NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

IMPLEMENTATION OF A HUMAN AVATAR FOR THE MARG PROJECT IN NETWORKED VIRTUAL ENVIRONMENTS

by

Faruk Yildiz

March 2004

Thesis Advisor:	Xiaoping Yun
Second Reader:	Don McGregor

Approved for public release; distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** March 2004	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: Implementation of a Human Avatar for the MARG Project in Networked Virtual Environments			**5. FUNDING NUMBERS**
6. AUTHOR(S) Faruk Yildiz			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) U.S. Army Research Office (ARO) and U.S. Navy Modeling and Simulation Office (N6M)			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release: distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

The objective of the ongoing MARG project is to animate human motions captured by 15 MARG sensors in wireless networked virtual environment (NVES). Three avatars were developed previously, but none of them met all the desired requirements. The first one was overly simplistic and did not implement H-Anim standards. The other two were created using laser-scanned data and followed the H-Anim standards, but one had its adjacent joints broken and the other was capable of rotating only one joint. Therefore, the cartoon-type humanoid, Andy, was developed to meet the needs of the MARG project. The humanoid Andy implements H-Anim standards using built-in X3D humanoid nodes and is capable of controlling all its 15 joints in NVES.

Another need of the MARG project was a wireless network interface for real-time data streaming. A concurrent client-server program implementing multicasting using TCP and UDP protocols was developed for this purpose. Using WiSER2400.IP serial adapters between the MARG sensors and the server program adds a wireless capability to the project. The server program converts the raw MARG sensor data to quaternions using the Quest algorithm. Multiple clients are supported by the system. Each client program receives the motion data and updates the humanoid Andy.

14. SUBJECT TERMS VRML, X3D, Java Network, Java, MARG sensor, Networked Virtual Environments, Virtual Environments, Humanoid, Avatar, Human Animation, Body Tracking, H-Anim, Control Interface Unit, WiSER2400.IP			**15. NUMBER OF PAGES** 78
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

This thesis is done in cooperation with the MOVES Institute
Approved for public release; distribution is unlimited.

IMPLEMENTATION OF A HUMAN AVATAR FOR THE MARG PROJECT IN NETWORKED VIRTUAL ENVIRONMENTS

Faruk Yildiz
Lieutenant Junior Grade, Turkish Navy
B.S.C.E., Turkish Naval Academy, 1998

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN
MODELING, VIRTUAL ENVIRONMENTS AND SIMULATION (MOVES)

from the

NAVAL POSTGRADUATE SCHOOL
March 2004

Author: Faruk Yildiz

Approved by: Xiaoping Yun
Thesis Advisor

Don McGregor
Second Reader

Rudolph P. Darken
Chair, MOVES Academic Committee

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The objective of the ongoing MARG project is to animate human motions captured by 15 MARG sensors in wireless networked virtual environment (NVES). Three avatars were developed previously, but none of them met all the desired requirements. The first one was overly simplistic and did not implement H-Anim standards. The other two were created using laser-scanned data and followed the H-Anim standards, but one had its adjacent joints broken and the other was capable of rotating only one joint. Therefore, the cartoon-type humanoid, Andy, was developed to meet the needs of the MARG project. The humanoid Andy implements H-Anim standards using built-in X3D humanoid nodes and is capable of controlling all its 15 joints in NVES.

Another need of the MARG project was a wireless network interface for real-time data streaming. A concurrent client-server program implementing multicasting using TCP and UDP protocols was developed for this purpose. Using WiSER2400.IP serial adapters between the MARG sensors and the server program adds a wireless capability to the project. The server program converts the raw MARG sensor data to quaternions using the Quest algorithm. Multiple clients are supported by the system. Each client program receives the motion data and updates the humanoid Andy.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION 1
- A. THE ONGOING MARG PROJECT 1
- B. THESIS GOALS 4
- C. ORGANIZATION 5
- D. SUMMARY 6

II. BACKGROUND 7
- A. NETWORKING PROTOCOLS 7
- B. PROGRAMMING LANGUAGES FOR NETWORKING (JAVA VERSUS C++) 9
- C. WIRELESS LAN 10
- D. 802.11B WIRELESS SERIAL PORT ADAPTER WISER2400.IP 10
- E. MAGNETIC, ANGULAR RATE, AND GRAVITY (MARG) SENSOR 12
- F. THE COMPLETE MOTION TRACKING SYSTEM 13
- G. SUMMARY 15

III. HUMANOID ANDY USING H-ANIM STANDARDS 17
- A. H-ANIM STANDARDS AND X3D GRAPHICS LANGUAGE 17
- B. FUNDAMENTAL H-ANIM NODES PROVIDED BY X3D 18
  - 1. Humanoid Node 18
  - 2. Joint Node 18
  - 3. Segment Node 19
  - 4. Displacer and Site Nodes 20
- C. HUMANOID ANDY 20
  - 1. The Nested Skeleton Structure 21
  - 2. The Rebuilding Process of Humanoid Andy 22
  - 3. Getting Humanoid Andy to Work with Java Networking 28
  - 4. Implementing Multiple MARG Sensors to Track Humanoid Andy 31
- D. SUMMARY 33

IV. DESIGN OF THE CLIENT-SERVER PROGRAM 35
- A. MULTICASTING USING TCP AND UDP PROTOCOLS 36
- B. IMPLEMENTATION OF MUTUP IN THE MARG PROJECT 40
- C. SUMMARY 45

V. TESTING AND EVALUATION 47
- A. ROTATIONAL MOTION TESTS THE HUMANOID ANDY 47
- B. TESTING HUMANOID ANDY WITH ONE MARG SENSOR 49
- C. TESTING HUMANOID ANDY USING TWO MARG SENSORS 51
- D. TESTING THE CONCURRENT CLIENT-SERVER PROGRAM 53
- E. FINAL RESULTS 54

VI. CONCLUSIONS AND FUTURE WORK 57
A. SUMMARY AND CONCLUSIONS 57
B. FUTURE WORK 59
LIST OF REFERENCES 61
INITIAL DISTRIBUTION LIST 63

LIST OF FIGURES

Figure 1. The Humanoid as a Boxman. [From Ref. 2.] 2
Figure 2. Dutton's Humanoid with Skin Deformations. [From Ref. 4.] 3
Figure 3. Sinav's Humanoid Implementing Deformation Engine. [From Ref. 5.] 4
Figure 4. Setting of WiSER2400.IP. 11
Figure 5. The Schematic Diagram of the MARG Motion. 14
Figure 6. The Components of the MARG Motion Tracking System. 15
Figure 7. Humanoid Node. [From Ref. 14.] 19
Figure 8. Joint Node. [From Ref. 14.] 20
Figure 9. Segment Node. [From Ref. 14.] 20
Figure 10. Detailed Skeleton Structure of a Humanoid. [From Ref. 1.] 23
Figure 11. The Skeleton of Humanoid Andy in Nested-Joint Structure. 24
Figure 12. The Pseudo Code for Recalculating the Vertexes. 25
Figure 13. The Meaning of the Ratio Value. 26
Figure 14. The Possible Locations on the X-Axis. 27
Figure 15. Rebuilding the New Humanoid in the Nested Joint Structure. 29
Figure 16. Humanoid Andy. 29
Figure 17. The Relationship between Java and X3D. 31
Figure 18. The Java Script Code. 31
Figure 19. The X3D Script Node. 32
Figure 20. Positioning One Segment. 33
Figure 21. Positioning Two Adjacent Segments in the Nested Structure. 34
Figure 22. Pseudo Code for the Server Class. 37
Figure 23. Pseudo Code for the HandleClient Class. 38
Figure 24. Pseudo Code for the Client Class. 39
Figure 25. Pseudo Code for the MemoryUpdater Class. 40
Figure 26. Block Diagram of Concurrent Client Server Program. 43
Figure 27. A Detailed Block Diagram of MemoryUpdater Class. 44
Figure 28. Simultaneous Rotation of the Left Forearm about Z-Axis and Left Hand about Y-Axis 48
Figure 29. Rotations of Left Forearm, Right Forearm and Head of the Humanoid Andy 48
Figure 30. A Sample Screen Capture of the LISP Program Moving an Arm in 3 DOF 49
Figure 31. The Implementation of Quest Algorithm to the Humanoid Andy (90 Degrees of Rotation about Negative X-Axis) 50
Figure 32. Testing Two MARG Sensors on the Humanoid Andy. 52
Figure 33. Testing Two MARG Sensors on the Humanoid Andy. 53

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I must first thank the U.S. Army Research office (ARO) and the U.S. Navy Modeling and Simulation office (N6M) for funding this research. Without their support, we would not have had the necessary means to fulfill our goals, and this research would never been completed.

I would also like to thank Professor Yun for everything he taught me during this research. It was an honor to work with such a distinguished educator. In addition, I thank Don McGregor for his guidance in this research. I also thank Andreas Kavousanos-Kavousanakis for the joint work with him, for getting the MARG sensors working, and for implementing the Quest algorithm.

I am also indebted to Professor McGhee for his guidance and for providing the LISP code to develop the program that produced the simulation motion data I used in my testing. I would also like to thank Ron Russell for editing this thesis in a timely and professional fashion.

Most of all, I must thank my loving and beloved wife, Ebru, for all her patience and support as I conducted the research for this thesis. With all my love, I dedicate this thesis to Ebru and to my beautiful newly born baby son, Uzeyir Alper. What a joy it was to experience the beginning of fatherhood while satisfying the rigors of this thesis.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

As the 3D graphics and the Internet continue to grow, the need to characterize human beings in networked virtual environment (NVE) will continue to increase. Creating and animating humanoids for different applications require establishing libraries of interchangeable humanoids and standardization for humanoids [Ref. 1]. Furthermore, because humanoids in a networked virtual environment will most likely run in computers with different architectures, both humanoids and other interfaces such as networking programs should be capable of running in multiple platforms. Java as a programming language and X3D as a 3D graphics tool met the needs described above. X3D has built-in nodes implementing H-Anim 2.0 specifications.

To animate humanoids in the virtual environment, one must acquire or generate motion data of human body limbs. Bachmann [Ref. 2] introduced Magnetic, Angular Rate, and Gravity (MARG) sensors, which use inertial/magnetic measurements for real-time human body-motion tracking. This new technology overcomes the limitations of previous motion-tracking technologies, and it is capable of tracking multiple users in wide areas. Since the current implementation of the MARG sensors has a drawback of limiting the users' ability and flexibility of movement in a tracking system, the wireless serial adapter WiSER2400.IP technology is used in this thesis for wirelessly delivering UDP packets to the network.

This chapter briefly discusses the ongoing MARG project and the novelties that this thesis adds. Additionally, an outline of the remaining thesis chapters is presented.

A. THE ONGOING MARG PROJECT

Bachmann [Ref. 2] successfully added humans into networked virtual environments by using MARG sensors. The motion of the human was accurately tracked with a 100 Hz update rate. The experimental results showed that inertial/magnetic orientation estimation is a practical method of tracking human body posture [Ref. 2]. However, transmitting sensor data by wires in the system was one of the serious drawbacks in this implementation. The ability to track human motion was restricted to

short distance because the sensors were directly wired to the desktop computer where the avatar was running. Another drawback with this implementation was that the humanoid used for representing the tracked motion was not realistic. Instead, as shown in Figure 1, a simple box man was implemented to represent the human being tracked [Ref. 2]. Furthermore, the skeleton structure was not developed using H-Anim standards. H-Anim standards require a humanoid skeleton in a nested hierarchical joint structure.

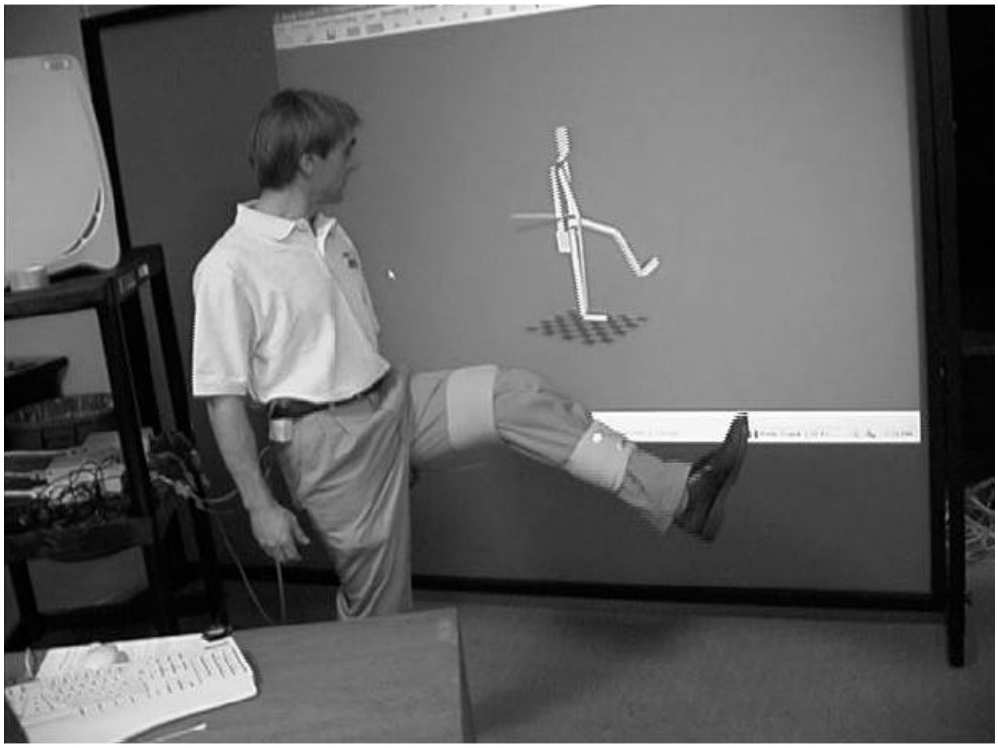


Figure 1. The Humanoid as a Boxman. [From Ref. 2.]

Dutton [Ref. 3] developed a more realistic humanoid by using laser-scan data. The data was parsed into segments and the humanoid was constructed using the Virtual Reality Modeling Language (VRML) by following the H-Anim 1.1 specifications. As illustrated in Figure 2, the major drawback with Dutton's humanoid was that the connections of two adjacent joints were broken when animating the humanoid [Ref. 3].

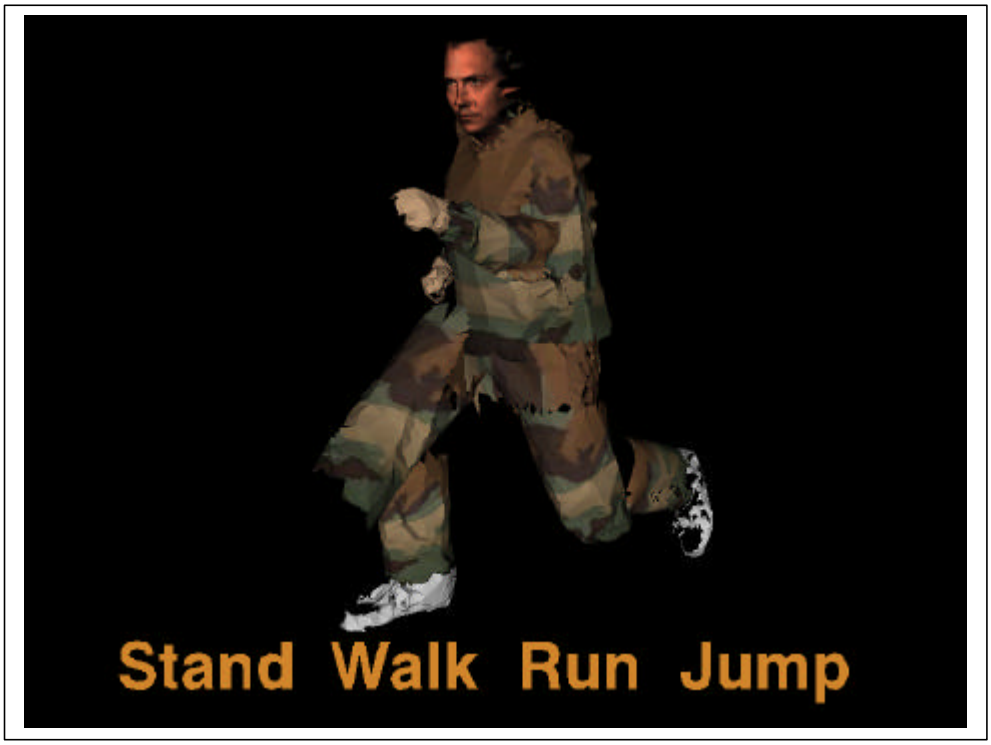


Figure 2. Dutton's Humanoid with Skin Deformations. [From Ref. 4.]

Sinav [Ref. 5] improved Dutton's humanoid and developed a deformation engine that eliminated skin deformation to a high degree and added smooth joint connections. Despite its smooth joint connections, Sinav's humanoid does have some geometric distortions of the segments when moving as shown in Figure 3. Another drawback with his humanoid is that the user can only control the joint for the head. Therefore, a desirable humanoid for representing the 15 MARG sensor data motion was still unavailable for the MARG project.

Developing a realistic humanoid is only one part of the ongoing MARG project for representing the tracked human motion. Another part of the project is developing a concurrent client-server program for implementing MARG project in a real-time networked virtual environment. Dutton [Ref. 3] wrote several java classes for reading and parsing pre-recorded body motion data from a file, wrapping the data into a UDP packet and sending it to the client program running the avatar over the network. These classes

were simply simulating the networked environment and were not feeding the humanoid in real-time tracked motion data. Furthermore, the server program was not capable of serving multiple clients simultaneously. The server program was supposed to receive MARG sensors data wirelessly through the network and deliver those data to the clients anywhere on the Internet. The client program should be capable of running a humanoid with at least 15 joints simultaneously updating motion data.

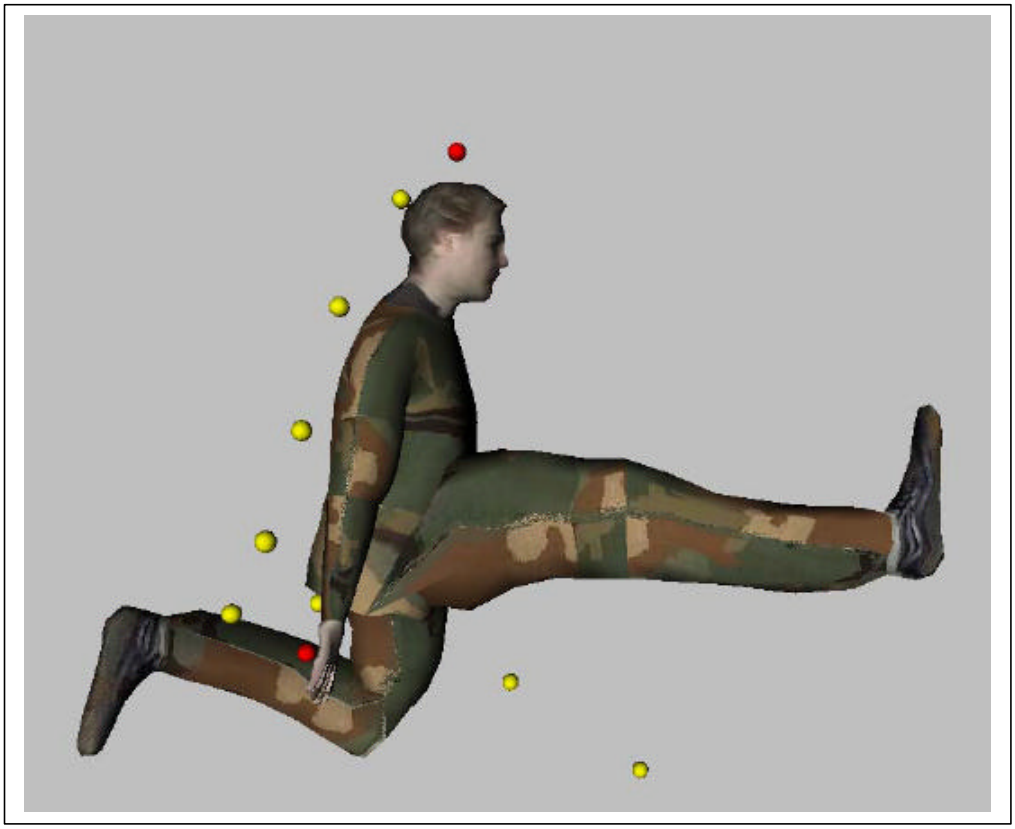


Figure 3. Sinav's Humanoid Implementing Deformation Engine. [From Ref. 5.]

B. THESIS GOALS

This thesis concentrates on developing a real-time implementation of human avatars animating the tracked human motion in a networked virtual environment. First, a new humanoid for the MARG project will be developed. This humanoid will be developed in a hierarchical joint skeleton structure using H-Anim200x nodes provided by X3D. It will have a resolution of at least 15 joints in order to animate motions of major

limbs of a human successfully [Ref. 2]. The appearance of the humanoid is not important for this thesis. A humanoid better than the Boxman but simpler than the previously developed laser-scanned humanoids will be sufficient. After searching the literature and Internet, a cartoon-type humanoid AndyLow from [Ref. 6] is chosen as the starting point.

Second, a concurrent client-server networking interface between the tracked human and multiple clients running anywhere on the Internet will be developed. This concurrent client-server application receives data from MARG sensors through the WiSER2400.IP wireless serial adapter, capable of handling multiple clients, each running a humanoid representing the same motion captured by the sensors. Since the 16-channel Control Interface Unit (CIU) is not ready at this moment, the three-channel CIU is used for connecting the sensors with the wireless serial adapters. The server program runs a class for producing quaternion data using the Quest Quaternion Algorithm [Ref. 7]. A filtering process is implemented in the sensor data before the algorithm is conducted. On the client side, the client program converts the sensor quaternion data, independent from each other, into a nested joint structure to make them compatible with H-Anim standard humanoid structure. An additional conversion from quaternion into axis-angle pair is conducted before updating the humanoid.

C. ORGANIZATION

This thesis contains six chapters. Chapter II presents background information for this thesis. It introduces networking protocols, compares Java and C as a networking programming language, describes wireless LAN, introduces MARG sensors and WiSER2400IP serial adapters, and provides the complete setting of real-time wirelessly networked, full-body tracking system using MARG sensors and WiSER2400.IP wireless serial adapters.

Chapter III introduces Virtual Reality Modeling Language (VRML), X3D and H-Anim 2.0 specification, explains how the nodes in X3D create a humanoid, discusses how the humanoid Andy is modified from humanoid AndyLow, explains how Java networking works with X3D and how independent sensor quaternion data is implemented to a nested-joint structure humanoid.

Chapter IV analyses the design of the concurrent client-server program and its purpose.

Chapter V describes the testing and evaluation of the concurrent client-server program and the client programs running the new humanoid. The one-channel and the three-channel CIUs developed by Kavousanos-Kavousanakis [Ref. 8] were used to obtain adjacent joint data for animation.

The final chapter presents the conclusions and suggests further development and optimization.

D. SUMMARY

This chapter discussed the existing state of the MARG project prior to this research. The early design of the humanoids did not meet the needs for this thesis. Moreover, a wireless networking interface has to be added.

II. BACKGROUND

Networking in virtual environments is becoming more popular each day. Combat models, games, virtual conferences are some of the applications that presently use networked virtual environments (NVES). Programming languages such as C++ and Java provide users high-level tools that eliminate the need to work with the complex low-level basics of network programming. The WiSER2400.IP wireless serial adapter [Ref. 9] is capable of receiving byte level data from a serial input and packing them either into a UDP/IP or TCP/IP packet. Then the packets are transmitted over a wireless link to the host IP address and port number set by the user. By connecting the MARG sensors [Ref. 2] to the WiSER2400IP device through the serial port, one can track real-time human motion in a wireless networked virtual environment. This chapter introduces networking protocols, compares Java and C as a networking programming language, describes wireless LAN, introduces MARG sensors and WiSER2400IP serial adapters, and finally provides the complete setting of a real-time wirelessly networked, full-body tracking system, using MARG sensors and WiSER2400IP devices.

A. NETWORKING PROTOCOLS

Most of today's Internet services are based on one sender and one receiver. The file transport protocol (FTP), hyperlink transfer protocol (HTTP) and simple mail transfer protocol (SMTP) are only some of the examples. Applications like video conferencing and audio streaming that have one sender and multiple receivers are less common because of the high bandwidth requirements. Depending on the number of senders and receivers and the type of the packet used for communication, there are three types of network communications: unicast, broadcast and multicast.

Unicast is the communication between one source host and one destination host. It is subdivided into two protocols, TCP and UDP. TCP/IP protocol provides a reliable data transmission between the source and the destination. The packets are delivered reliably in order and checksums are implemented to packets to avoid data transmission errors. Additionally, transmission flow is controlled to prevent the congestion on the

network. All these services of course have a cost of delay. FTP, HTTP and SMTP are examples of unicast TCP protocols. On the other hand the UDP/IP protocol removes most of the communication overhead used by TCP protocol. It is based on a simple structure offering best-effort packet data delivery. Reliability and the order of the packets are not guaranteed. It is well suited for applications such as video streaming because these application classes depend on low latency and jitter.

Broadcast is the communication between one source host and all the other destinations hosts on a subnet. It is largely parallel with UDP/IP protocol with the exception that the packets are delivered to all hosts on a local network. Applications such as address-resolution protocol (ARP) use broadcasting to map the IP network addresses to the hardware addresses (MAC addresses) in the data-link layer.

Multicasting is the communication between one or more source hosts and a group of receiver hosts located anywhere on the network. Multicasting is more efficient than unicast or broadcast for transmitting information among a large number of group members spread out over different networks. Multicast routing algorithms can distribute data across network boundaries, unlike broadcast, while not sending duplicate copies of data, as with unicast. Packets are restricted by the multicast routing algorithms to travel only along networks that have subscribers to that group and to newer travel over a single network multiple times. A new multicast group member must send a join message, which is distributed to the other routers participating in the multicast group distribution.

In many cases, multicasting capability is desirable. The major advantage of multicasting is that it reduces the use of network bandwidth. Assuming that there are 100 members of a group, transmitting only one packet by the source will be sufficient in multicasting, while 100 packets are required to be sent separately to each group member with unicast UDP/IP.

Despite the advantages listed above, multicasting has the following disadvantages. The majority of routers on the Internet today are not configured to handle multicasting routing protocols. Most exclusively handle unicast or broadcast traffic. A virtual network using *tunneling* can be used with unicast-only routers to overcome this problem. Tunneling is a software solution that runs on the end point routers/computers

and allows multicast packets to traverse the network by putting multicast packets into unicast packets.

Programming languages compatible with networking protocols provide socket classes to represent the terminals of a connection between two machines or processes. For instance, the java.net package provided by Java contains a Socket class for TCP connection, a DatagramSocket class for UDP connection and a MulticastSocket class for a multicast connection.

Although the IP address is unique to each computer, it is insufficient to differentiate between multiple applications running on the same computer. Packets that arrive at the machine must know which application they should be processed by. This is accomplished through the use of port numbers. The port number is represented by a 16-bit unsigned number that has a range from 0 to 65,536. Both the TCP and the UDP have their own port numbers in the same range. Depending on the platform, port numbers are divided into three ranges: the well-known ports (from 0 through 1023), the registered ports (from 1024 through 49151), and the dynamic ports (from 49152 through 65535). The well-known ports are assigned by the Internet Assigned Numbers Authority (IANA) for special usage, such as 21 for FTP, 23 for Telnet, 25 for SMTP (mail) and 80 for HTTP (Web).

B. PROGRAMMING LANGUAGES FOR NETWORKING (JAVA VERSUS C++)

Java, compared to C++, is a painless networking language. Some of the advantages of Java are as follows. First, Java supports threads at the language level. C++ supports threads, but it is complicated to program and varies from platform to platform. Developing a program with concurrent processing is difficult, if not impossible, without threads. The concurrent processing is vital in applications like File Transport Protocol. Second, Java has the advantage of being portable between systems, even without recompilation. It is easy to access machine level details in C++, but this makes C++ depend on to the specific platform on which the program was implemented. Considering the number of different kind of machines connected to the Internet, it is obvious that portability feature adds enormous power to Java. Finally, Java is much easier to use.

Many of the details require to set up a network connection are hidden by abstraction. Furthermore, the garbage collection feature of Java handles all undeleted objects and frees the memory. The most important disadvantage of Java is the speed. C is very successful in computation-intensive applications. [Ref. 10]

C. WIRELESS LAN

A wireless Local Area Network (LAN) is designed to transmit and to receive data over the air to minimize the need for wires in communication. The wireless communication can be established either between a wireless client and a base station or between two wireless stations. The standard for wireless LAN is referred as IEEE 802.11 established in 1997 [Ref. 11]. Since 2.4 GHz is an unlicensed frequency band in most countries, using this frequency band for data transmission makes IEEE 802.11 a global standard.

The current IEEE 802.11 technology consists of four different types: 802.11, 802.11a, 802.11b, 802.11g. The 802.11 standard offers a 1 Mbps or 2 Mbps transmission rate in the 2.4 GHz band. The 802.11a standard provides a 54 Mbps transmission rate in a 5 GHz band. The 802.11b standard is a high rate (or Wi-Fi) extension of the 802.11 and provides up to an 11 Mbps transmission rate in a 2.4 GHz band. It is also backward compatible with 5.5 Mbps, 2 Mbps and 1 Mbps rates. Finally, the 802.11g is the latest standard that offers up to 54 Mbps in the 2.4 GHz band. [Ref. 12]

D. 802.11B WIRELESS SERIAL PORT ADAPTER WISER2400.IP

The WiSER2400.IP is an 802.11b compliant module with an RS232 serial interface. The WiSER2400.IP takes the serial data from the equipment it is connected to, via the RS-232 serial port, and transmits them to the destination host. The data is encapsulated into either TCP/IP or UDP/IP packets to make WiSER2400.IP compatible with networking protocols. This capability of WiSER2400.IP is very useful in networking applications, so it is very handy to establish a communication between the WiSER2400.IP and the remote applications. [Ref. 9,13]

In order to use WiSER2400.IP serial adapter in this thesis research, the following settings are configured via the utility program (wauti.exe) provided by the manufacturer. A screenshot of the configuration is provided in Figure 4. After connecting the WiSER2400.IP to the serial port and selecting the port name in the utility program, clicking the detect button detects the adapter and makes it ready for the settings. Once the adapter is detected, the TCP/UDP radio button is selected as a first step to establish a TCP or UDP communication with the destination application. As a second step, the wireless settings are set. WiSER2400.IP can operate in two modes: Ad-Hoc and infrastructure mode. In Ad-Hoc mode, computers can talk directly to each other and do not need an access point. But, in this thesis WiSER2400.IP is talking to the computers through an access point, and therefore infrastructure mode is selected as the network type. Additionally, SSID is set to the same SSID used by the access point. The third step

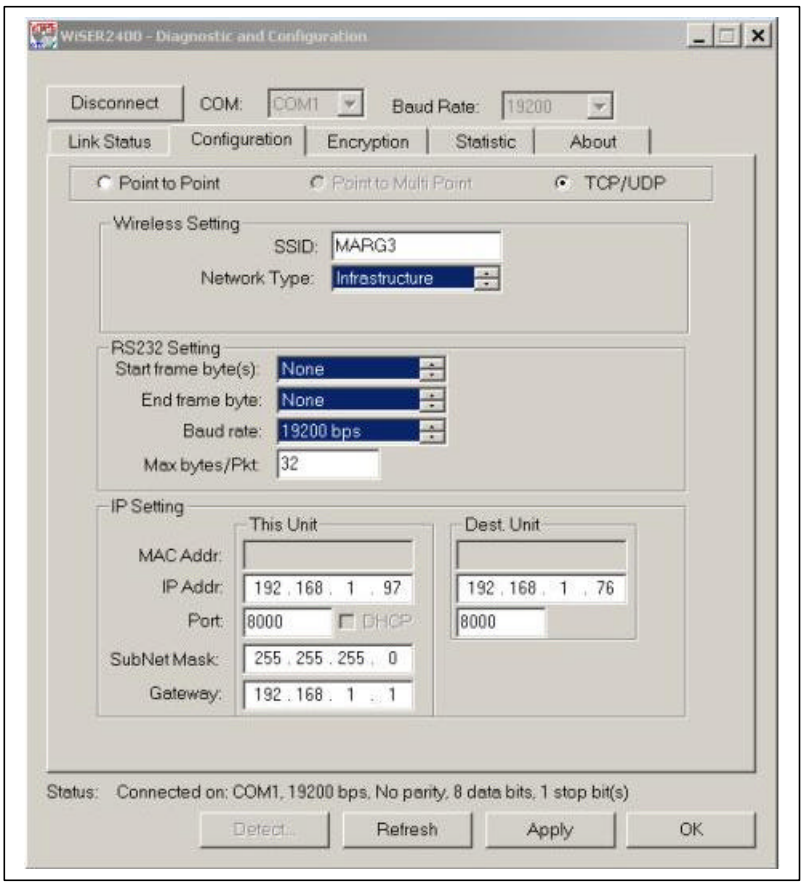


Figure 4. Setting of WiSER2400.IP.

is to set the RS232 settings. The baud rate used in this thesis is 19200 bps. Because a sample motion data consists of 15.5 bytes, the packet size is set as twice of a sample data size. The maximum packet size can be set as 200 bytes, which is insufficient for further work with 15 MARG sensors. However, the current packet size is sufficient for this thesis because only one sensor is connected to the serial adapter. Finally the IP settings should be configured correctly. The destination unit's IP address and port number indicate the location of the application that listens for the packets from WiSER2400.IP. In addition to setting the destination address and port number for the UDP/IP communication, the user needs to set the unit's IP address, port number, subnet mask and gateway information, too.

E. MAGNETIC, ANGULAR RATE, AND GRAVITY (MARG) SENSOR

All materials in this section are drawn from [Ref. 2]. The power of Networked Virtual Environments lies in its ability to immerse users in a different world. The more complete the immersion, the better and more effective the virtual environments (VE). If user interactions are done with VE in the same manner as they are done in the real world, they increase the immersion. The human interactions occur as a result of body motion. Many different types of motion-tracking sensors including mechanical, optical, acoustic and magnetic trackers have been introduced. Each of these tracker technologies has limitations including marginal accuracy, user encumbrance, restricted range, susceptibility to interference and noise, poor registration, occlusion difficulties and high latency. These limitations make it difficult to track multiple users in virtual environments and augmented reality applications. The MARG sensor introduced by Bachmann [Ref. 2] overcomes the limitations of the motion-tracking technologies above.

The MARG motion-tracking sensor is a new sourceless tracking technology for tracking the posture of an articulated rigid body. Source-based tracking systems require a continuous link between the tracked body and one or more fixed stations. Since the distance that can be maintained by the link when wired technology is used is often limited, MARG sensors offer users an enormous distance that is limited with the wireless technology. MARG technology is based on the use of inertial/magnetic sensors to determine the orientation of each link in the rigid body independently. Each sensor

produces nine components of data (three rate sensor measurements, three accelerometer measurements and three magnetometer measurements) for the tracked motion in a 100-Hz update rate.

F. THE COMPLETE MOTION TRACKING SYSTEM

Although the goal of this thesis is to develop an avatar to be used with a 15-channel Control Interface Unit (CIU), the avatar is tested with the only available 3-channel CIU. Figure 5 shows a schematic diagram of the MARG motion tracking system using the 3-channel CIU and three WiSER2400.IP serial adapters. Each channel in the CIU is independent of the other. That is, the user can use one sensor at a time, or any possible combinations of the three. The data tracked by the first sensor is delivered to the WiSER2400.IP through the first input and output port of the CIU. The WiSER2400.IP accepts data only via the RS232 serial port interface.

Each serial adapter must have the same destination IP address but may have a different destination port number. The destination IP address is the IP address of the host where the server program runs. Each serial adapter uses a different port number because each is handled independently of the other in the server program. The port numbers used in this thesis are UDP 8000, 8001 and 8002. There will be only one serial adapter when the 15-channel CIU is available in further work. In that case, the CIU will have a 16-input channel and one-output channel. All sensors will be connected to the CIU and the CIU will serialize the read data and forward to the serial adapter. There will be only one serial adapter in the system and additional UDP ports will not be needed.

Because a wireless access point is connected to the LAN, the wireless network type specification of each WiSER2400.IP is set at infrastructure mode. When the IP address of the server is set correctly in the serial adapters, the computer that runs the server program can be either in the same LAN as the wireless access point or in any other network on the Internet. It is the same concept for the computers that run the client program. The components of the MARG motion tracking system are shown in Figure 6.

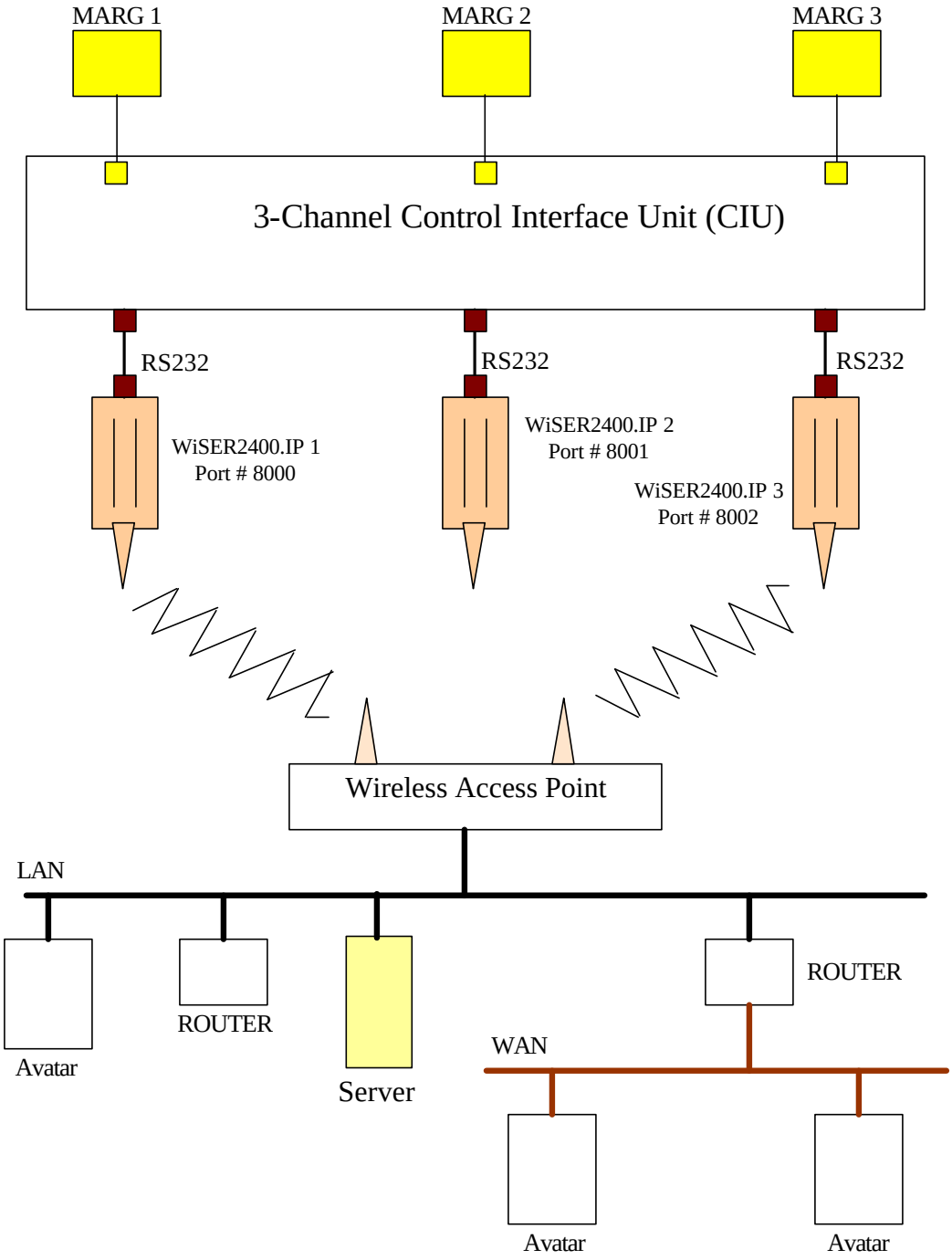
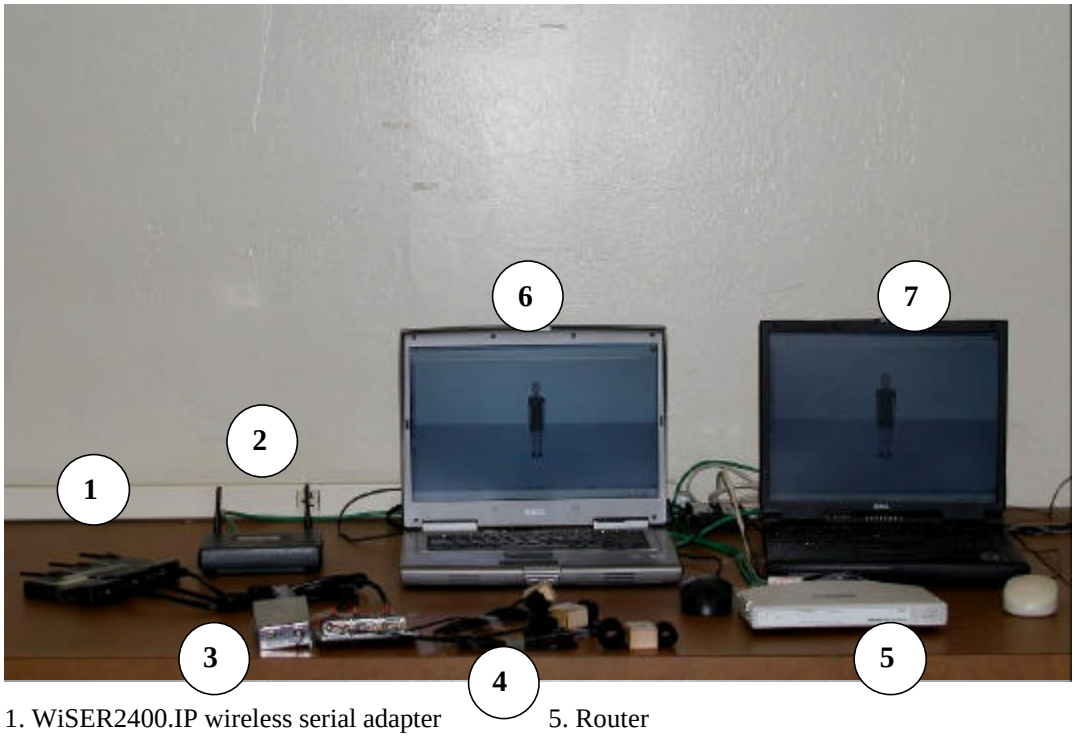


Figure 5. The Schematic Diagram of the MARG Motion.



1. WiSER2400.IP wireless serial adapter
2. Wireless Access Point
3. Three-channel CIU
4. MARG Sensors
5. Router
6. PC running server program and a client
7. PC running a client

Figure 6. The Components of the MARG Motion Tracking System.

G. SUMMARY

Java as a painless networking tool makes it easier to develop concurrent programs, which is vital for this thesis. Various classes contained in java.net package make network programming very easy. Its portability between systems adds noteworthy strength to Java in the Internet world.

Unicasting, broadcasting and multicasting are the major types of communications used in networking programming. Unicasting is sub-divided into TCP and UDP protocols. TCP provides a reliable but higher latency and higher jitter communications while UDP provides less reliable but lower latency and lower jitter communications. Because of its low latency, UDP is preferred in streaming application such as the video, voice or real-time applications discussed in this thesis research.

Adding the WiSER2400.IP serial adapter for this thesis makes it possible to track human motion in a wireless network environment. However, using three different wireless serial adapters with the 3-channel CIU seems to limit human motions. The further version of CIU with 15 channels will eliminate extra WiSER2400.IP usage. The 15-channel CIU packs all the data for 15 sensors into the serial output.

III. HUMANOID ANDY USING H-ANIM STANDARDS

This chapter first presents the H-Anim standards and the X3D graphics language. Second, the H-Anim nodes contained by X3D and the nested joint structure of humanoids are described. Third, the method for feeding the nested joint structure humanoid with multiple MARG sensor data is discussed. And finally, the required settings for updating the humanoid through the network are explained.

A. H-ANIM STANDARDS AND X3D GRAPHICS LANGUAGE

The need for humanoids in the networked virtual environment continues to increase with the growth of the Internet. Libraries of humanoids using international standards are developed in order to meet this increasing need. Human Animation (H-Anim) is a newly developed standard for this purpose. This standard is not language specific and can be applied to any applications using any programming languages.

The joint structure of the humanoid offered by H-Anim is a nested joint structure. That is, the child joints depend on their parent joints. For example, the left shoulder joint is the parent joint to the left elbow joint. The motion applied to the left shoulder is automatically applied to the left elbow. For this reason, the animation for two adjacent joints will not work for two MARG sensors when the captured motion data is applied directly. This is because each MARG sensor works independently of the others, and a computation is needed before applying the data to the humanoid.

Virtual Reality Modeling Language (VRML) is a 3D graphics language for developing models used in virtual environment applications. Extensible 3D (X3D) is an extension of VRML and contains built-in nodes implementing H-Anim standards. X3D is also compatible with the Java programming language, and Java Script classes can be embedded into humanoids through the Script nodes provided by X3D. This makes it possible to update humanoids with the data received through the network. Unfortunately, X3D is not presently compatible with Java 1.4 or later versions in network applications. This issue will be discussed later in this chapter.

B. FUNDAMENTAL H-ANIM NODES PROVIDED BY X3D

Humanoid, Joint, Segment, Site and Displacer nodes are the fundamental nodes provided by X3D to support H-Anim specifications. These nodes are used to construct the nested joint structure of the humanoid. This thesis uses only the Humanoid, Joint and Segment nodes for creating the humanoid. Shape, Transform, Viewpoint and Color are some additional nodes of X3D used for defining the geometry of the human limb segments. These nodes are not used in this thesis and will not be discussed. All materials in this section are drawn from [Ref. 14].

1. Humanoid Node

Humanoid node is a container node for Joint, Segment, Site, and Displacer nodes. It also contains the geometry nodes Shape, Transform and Color, as well as a Viewpoint node. The author and copyright information of the humanoid is kept in this node.

The translation and the rotation fields specify a translation or a rotation to the coordinate system of the entire humanoid figure. The scale field specifies a non-uniform scale of the humanoid figure coordinate system and the scale values must be assigned greater than zero. The Viewpoint node is affected by the transformations and rotations applied to the Humanoid node, but not affected by any of the transformations performed to the Joint nodes. The structure of Humanoid node is provided in Figure 7.

2. Joint Node

Joint nodes represent the joints in the body. The function of a Joint node is to define the relationship of two adjacent segments. An organization of hierarchical Joint nodes describes the overall skeleton of the humanoid in a nested joint structure. A Joint node can be a child of another Joint node or the Humanoid node, but it cannot be a child of a Segment node. A Joint node can contain Segment nodes.

Joint nodes are the nodes used for animating the humanoids. The rotation field determines the posture of the joint. Since joints are in a nested structure, the amount of rotation set to this field is relative to the parent node. That is, the parent joint initially sets its rotations and then the child joint sets its rotation according to the final local coordinate

system achieved by the parent joint. Because of this reason, the measured motion data by the MARG sensors cannot be directly implemented to the Joint nodes. Before setting each Joint node's rotation field, an inverse motion should be implemented in order to achieve the original coordinate system. Implementation of this method is discussed later in this chapter. It is possible to set limits to the rotation fields using limitOrientation field, but it is not implemented in this thesis.

```
Joint : X3DgroupingNode {
        field [ ]          SFVec3f        bboxCenter        # init val: 0 0 0
        field [ ]          SFVec3f        bboxSize          # init val: -1 -1 –1
        field [in, out]    SFVec3f        center            # init val: 0 0 0
        field [in, out]    MFNode         humanoidBody      # init val: [ ]
        field [in, out]    MFString       info              # init val: [ ]
        field [in, out]    MFNode         joints            # init val: [ ]
        field [in, out]    SFString       name              # init val: " "
        field [in, out]    SFRotation     rotation          # init val: 0 0 1 0
        field [in, out]    SFVec3f        scale             # init val: 1 1 1
        field [in, out]    SFRotation     scaleOrientation  # init val: 0 0 1 0
        field [in, out]    MFNode         segments          # init val: [ ]
        field [in, out]    MFNode         sites             # init val: [ ]
        field [in, out]    SFVec3f        translation       # init val: 0 0 0
        field [in, out]    SFString       version           # init val: "1.1"
        field [in, out]    MFNode         viewpoints        # init val: [ ]
}
```

Figure 7. Humanoid Node. [From Ref. 14.]

The translation field is used to determine the reference location of the Joint node to the parent Joint node. The structure of Joint node is provided in Figure 8.

3. Segment Node

Segment node represents each segment of the body, such as pelvis, thigh or calf. This specialized grouping node provides a container for nodes in its children field. The children field may contain nodes, such as Shape or Transform, for drawing the geometry of the segment. In order to avoid violation of the structure of the H-Anim specifications, Segment nodes are allowed only as a child to the Joint nodes. Because Segment nodes represent the geometry of a human limb segment, there is no field provided for controlling the motion. The structure of Segment node is provided in Figure 9.

```
Joint : X3DgroupingNode {
        field [in, out]   SFVec3f        center            # init val: 0 0 0
        field [in, out]   MFNode         children          # init val: [ ]
        field [in, out]   MFFloat        llimit            # init val: [ ]
        field [in, out]   SFRotation     limitOrientation  # init val: 0 0 1 0
        field [in, out]   SFString       name              # init val: “ ”
        field [in, out]   SFRotation     rotation          # init val: 0 0 1 0
        field [in, out]   SFVec3f        scale             # init val: 1 1 1
        field [in, out]   SFRotation     scaleOrientation  # init val: 0 0 1 0
        field [in, out]   MFFloat        stiffness         # init val: [0 0 0]
        field [in, out]   SFVec3f        translation       # init val: 0 0 0
        field [in, out]   MFFloat        ulimit            # init val: [ ]
}
```

Figure 8. Joint Node. [From Ref. 14.]

```
Segment : X3DgroupingNode {
        field []          SFVec3f        bboxCenter        # init val: 0 0 0
        field []          SFVec3f        bboxSize          # init val: -1 -1 -1
        field [in, out]   SFVec3f        centerOfMass      # init val: 0 0 0
        field [in, out]   MFNode         children          # init val: [ ]
        field [in, out]   SFNode         coord             # init val: NULL
        field [in, out]   MFNode         displacers        # init val: [ ]
        field [in, out]   SFFloat        mass              # init val: 0
        field [in, out]   MFFloat        momentsOfInertia  # init val: [0 0 0 0 0 0 0 0 0]
        field [in, out]   SFString       name              # init val: “”
        event [in]        MFNode         addChildren
        event [in]        MFNode         removeChildren
}
```

Figure 9. Segment Node. [From Ref. 14.]

4. Displacer and Site Nodes

The displacer nodes are used to identify specific groups of vertices within a Segment node. Site nodes are used to define an attachment point for special accessories, such as nametags of companies or clothing. Another purpose of the Site node is to define a location for the viewpoints.

C. HUMANOID ANDY

Humanoid Andy is created using Extended 3D (X3D) language as a modification of the humanoid named AndyLow. AndyLow is developed as a low resolution humanoid

by Seamless Solutions, Inc. in 1998 [Ref. 6] and it is allowed to be used or modified for none commercial applications, provided that it carries the nametag of the company. AndyLow is selected as a starting humanoid for the humanoid Andy because it was based on H-Anim 1.1 standards and VRML97 (the former version of X3D). AndyLow cannot be used directly in this thesis because of its two drawbacks. First, the humanoid nodes were declared as proto declarations because VRML97 does not support built-in H-Anim nodes. This drawback is not so vital, but humanoid nodes provided by X3D are very handy. Second, the vertexes for the geometries were defined according to a unique reference point. When applying rotation to the joints, the segments do not rotate about their connection point to the parent. Instead, they rotate about the global reference position. This drawback is the main reason to rebuild AndyLow as Andy.

1. The Nested Skeleton Structure

Two different methods, according to the usage of reference position while creating the geometry of the human limb segments, can be implemented to represent a human in 3D-graphics world. The first method uses a fixed reference position. The vertexes of the geometry for each segment are determined using this reference position. When trying to implement rotations to the joints in this method, rotations are done about the fixed reference position. In the second method, each segment has its own local reference position defined separately. Local reference positions are assumed to be the connection points for two adjacent segments. Therefore, the transformation process is applied to all joints to place them into the correct location in the humanoid while creating it. For animation, each joint is rotated about its local reference position, i.e. a connection point.

As stated before, a Segment node cannot be a container for Joint nodes. Joint nodes are the containers to construct the skeleton of the humanoid. The construction of the skeleton can be achieved in two ways: Independent joint structure and nested joint hierarchy. The joints in the independent joint structure do not follow the parent-child relationship. Joint nodes are translated to their location according to a unique reference location. The motion applied to a Joint node does not affect the other joints. Since a MARG sensor measures the motion of one human limb independently of each other, this

method may seem more suitable for this thesis. But to be compatible with the latest standards, this thesis uses the H-Anim standard that defines the structure of the human skeleton completely differently. Nested joint hierarchy defined by H-Anim follows the parent-child relationship between two adjacent joints. A change applied to the parent node effects all the sub nodes. That is, when the left shoulder (upper arm) is rotated, the left elbow (forearm) will be rotated, too.

Figure 10 shows the complete skeleton structure defined by H-Anim standards. The Humanoid node is the main container for the whole humanoid. The first-level child joint is the Joint node *hanim_HumanoidRoot* that is the parent for all the remaining nodes. The second-level joints are *vl5* and *sacroiliac*. Totally there are 94 joints in a detailed skeleton structure.

A detailed skeleton structure is not required in this thesis because the MARG sensors are still too large to mount onto small segments, and the CIU considered for the MARG project at the current state supports only up to 15 MARG sensors. For this reason, details such as fingers are eliminated and only the most required 15 joints are selected. As illustrated in Figure 11, humanoid Andy has only 15 joints in its skeleton structure and each joint is assigned a number. These numbers are used as a standard for this thesis. In addition to the numbering, the segments attached to the joints are also shown in this figure. For example, *l_thigh* (left thigh) is the segment connected to the *l_hip* (left hip) joint. Therefore, *l_thigh* segments motion will be controlled through *l_hip* joint.

2. The Rebuilding Process of Humanoid Andy

The humanoid AndyLow was created using VRML97 language, and the humanoids nodes were self defined Proto nodes. The X3D built-in humanoid nodes already included the functionalities of these Proto nodes. The Proto nodes contained the geometry for the human limb segments. The vertexes in these geometries are very hard to set manually. The first step of the rebuilding process is to obtain these geometries in X3D type nodes. VRML97 applications can be imported into a project or converted to a new project in X3D. This is done by selecting file/import/VRML97 from the menu of X3D.

```
HumanoidRoot : sacrum
 sacroiliac : pelvis
 |  l_hip : l_thigh
 |    l_knee : l_calf
 |      l_ankle : l_hindfoot
 |        l_subtalar : l_midproximal
 |          l_midtarsal : l_middistal
 |            l_metatarsal : l_forefoot
 |  r_hip : r_thigh
 |    r_knee : r_calf
 |      r_ankle : r_hindfoot
 |        r_subtalar :  r_midproximal
 |          r_midtarsal : r_middistal
 |            r_metatarsal : r_forefoot
 vl5  : l5
  vl4 : l4
   vl3  : l3
    vl2 : l2
     vl1  : l1
      vt12 : t12
       vt11 : t11
        vt10  : t10
         vt9 : t9
          vt8 : t8
           vt7 : t7
            vt6  : t6
             vt5 : t5
              vt4 : t4
               vt3 : t3
                vt2 : t2
                 vt1  : t1
                  vc7 : c7
                  | vc6 : c6
                  |  vc5 : c5
                  |   vc4  : c4
                  |    vc3 : c3
                  |     vc2  : c2
                  |      vc1 : c1
                  |       skullbase : skull
                  |        l_eyelid_joint : l_eyelid
                  |        r_eyelid_joint : r_eyelid
                  |        l_eyeball_joint : l_eyeball
                  |        r_eyeball_joint : r_eyeball
                  |        l_eyebrow_joint : l_eyebrow
                  |        r_eyebrow_joint : r_eyebrow
                  |        temporomandibular : jaw
                  l_sternoclavicular : l_clavicle
                  | l_acromioclavicular : l_scapula
                  |  l_shoulder : l_upperarm
                  |   l_elbow : l_forearm
```

```
                  |    l_wrist : l_hand
                  |     l_thumb1 : l_thumb_metacarpal
                  |      l_thumb2 : l_thumb_proximal
                  |       l_thumb3 : l_thumb_distal
                  |     l_index0 : l_index_metacarpal
                  |      l_index1 : l_index_proximal
                  |       l_index2 : l_index_middle
                  |        l_index3 : l_index_distal
                  |     l_middle0 : l_middle_metacarpal
                  |      l_middle1 : l_middle_proximal
                  |       l_middle2 : l_middle_middle
                  |        l_middle3 : l_middle_distal
                  |     l_ring0 : l_ring_metacarpal
                  |      l_ring1 : l_ring_proximal
                  |       l_ring2 l_ring_middle
                  |        l_ring3 : l_ring_distal
                  |     l_pinky0 : l_pinky_metacarpal
                  |      l_pinky1 : l_pinky_proximal
                  |       l_pinky2 : l_pinky_middle
                  |        l_pinky3 : l_pinky_distal
                  r_sternoclavicular : r_clavicle
                   r_acromioclavicular : r_scapula
                    r_shoulder : r_upperarm
                     r_elbow : r_forearm
                      r_wrist : r_hand
                       r_thumb1 : r_thumb_metacarpal
                        r_thumb2 : r_thumb_proximal
                         r_thumb3 : r_thumb_distal
                       r_index0 : r_index_metacarpal
                        r_index1 : r_index_proximal
                         r_index2 : r_index_middle
                          r_index3 : r_index_distal
                       r_middle0 : r_middle_metacarpal
                        r_middle1 : r_middle_proximal
                         r_middle2 : r_middle_middle
                          r_middle3 : r_middle_distal
                       r_ring0 : r_ring_metacarpal
                        r_ring1 : r_ring_proximal
                         r_ring2 : r_ring_middle
                          r_ring3 : r_ring_distal
                       r_pinky0 : r_pinky_metacarpal
                        r_pinky1 : r_pinky_proximal
                         r_pinky2 : r_pinky_middle
                          r_pinky3 : r_pinky_distal
```

Figure 10. Detailed Skeleton Structure of a Humanoid. [From Ref. 1.]

```
HumanoidRoot : sacrum/pelvis                [0]
  |
  | l_hip : l_thigh                           [1]
  |     l_knee : l_calf                       [2]
  |          l_ankle : l_hindfoot             [3]
  |
  | r_hip : r_thigh                           [4]
  |     r_knee : r_calf                       [5]
  |          r_ankle : r_hindfoot             [6]
  |
  | vl5   : l5                                [7]
        |
        | skullbase : skull                   [8]
        |
        | l_shoulder : l_upperarm             [9]
        |     l_elbow : l_forearm             [10]
        |          l_wrist : l_hand           [11]
        |               l_fingers : left_fingers   [None]
        |
        | r_shoulder : r_upperarm             [12]
        |     r_elbow : r_forearm             [13]
        |          r_wrist : r_hand           [14]
        |               r_fingers : reft_fingers   [None]
```

Figure 11. The Skeleton of Humanoid Andy in Nested-Joint Structure.

The conversion by itself is not enough because X3D does not know how to the Proto nodes are defined. So, the nested hierarchical skeleton structure is rebuild by using Humanoid, Joint and Segment nodes. The geometry is the same geometry used by the Proto nodes. The result of this step makes humanoid AndyLow compatible with X3D.

The vertexes of each segment's geometry are defined according to a unique reference position located in the origin of the humanoid. Using the same reference position for all segments has drawbacks when rotating the joints, as described earlier in this chapter. Therefore, the second step is to recalculate the vertexes according to their local reference positions. The recalculation process is implemented as follows: The original vertex values are copied from the geometry nodes and pasted into a file. PointsCalculator class reads the original vertex values from this input file and saves the recalculated new results into an output file. New vertexes in the output file is copied and

pasted back to the geometry nodes. This process is repeated for each segment. The pseudo code for the recalculation process is provided in Figure 12.

```
// Assign the RATIO values.
// Ration value is the ratio of length from the parent joint (local reference position).
// The lower the value, the local reference position (connection) closer to the parent joint
ratio_X = 0.5;     ratio_Y = 0.1;     ratio_Z = 0.5;

// Find the minimum and maximum vertex values for the given geometry (in 3 axis)
//  loop (until data left in input file) {
//          Read vertex value from file and compare with current min and max values  }
min_X, min_Y, min_Z, max_X, max_Y, max_Z

// Find the length of the geometry for 3 axes
// length = abs(max vertex value - min vertex value)
length_X, length_Y, length_Z

// Determine the local reference position of the geometry.
// Positioned from the closest end to the parent joint in the amount of RATIO value
// locationRefPos = length * RATIO
locationRefPos_X, locationRefPos_Y, locationRefPos_Z

// Determine the required shifting value to move the local reference position to the origin
// if the closest edge of the geometry to the origin is the MAX vertex value:
//                              shift = - (max value - locationRefPos)
// if the closest edge of the geometry to the origin is the MIN vertex value:
//                              shift = - (min value + locationRefPos)
shift_X, shift_Y, shift_Z

// save the shifting values to the output file

// apply these shifting values to the three axis of the vertex.
loop (until no data left in the input file) {
        // Read the next vertex from the file
        read from file

        // new value = old value + shift
        new_X, new_Y, new_Z

        // save the new vertex to the output file
        save to file
}
```

Figure 12. The Pseudo Code for Recalculating the Vertexes.

The ratio values in the vertex recalculation pseudo code determine the distance for the connection point of the geometry from the parent joint. For example, a ration value of 0.1 indicates the 10 percent length of the total geometry. The lower the ratio value, the closer the connection to the parent joint. The connection point is also referred to as the

local reference position for new vertexes. The default values are 0.5 for x-axis and z-axis, 0.1 or 0.9 for y-axis. Because, according to the standard posture for humans in H-Anim standard, a human's face is out to the positive z-axis, the head is to the positive y-axis, and the left side is to the positive x-axis in a standing position (attention position in military) [Ref. 1]. In a standing position, the segments are mostly postured downward. For example, the thickness of the forearm is represented by the x- and z-axis while the length is represented by the y-axis. Figure 13 illustrates the meaning of the ratio value.

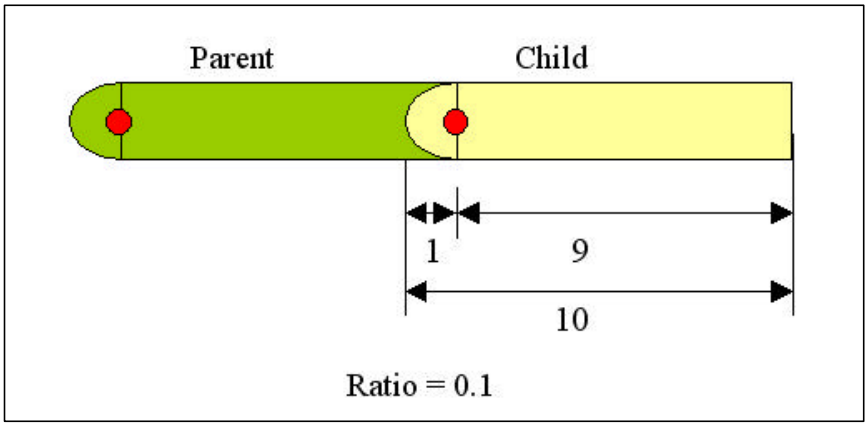


Figure 13. The Meaning of the Ratio Value.

Since the goal of this algorithm is to determine a connection point to the parent joint and redraw the complete geometry in the new reference position, the next step of the algorithm is to find a shifting vector from the local reference position to the origin. The maximum and minimum vertex values of the geometry in each axis are determined. Taking the absolute values of the differences between the maximum and minimum values give the lengths of the geometry in each axes. Multiplying the lengths with their corresponding ratio values will give the distance from the local reference position to the parent joint.

The shifting vector is the vector required to translate the connection point to the origin. It is determined using two different types of calculations depending on the position where the geometry is located. The possible four locations on the x-axis are illustrated in Figure 14. Positions in case 1 have their max-valued edge closer to the origin. The absolute maximum value is smaller than the absolute minimum value. That is,

most parts of the geometry (or the complete geometry) locate in the negative portion of the x-axis. The remaining possible locations in case 2 have their minimum valued edge closer to the origin. In this case, the absolute minimum value is smaller than the absolute maximum value. That is, most of the geometry (or the complete geometry) locates in the positive portion of the x-axis. The shifting value for x-axis is determined as follows:

shift value = - (max value – length * ratio) (in case 1)

shift value = - (min value + length * ratio) (in case 2)

The same concept is followed for the y- and z-axis, too.

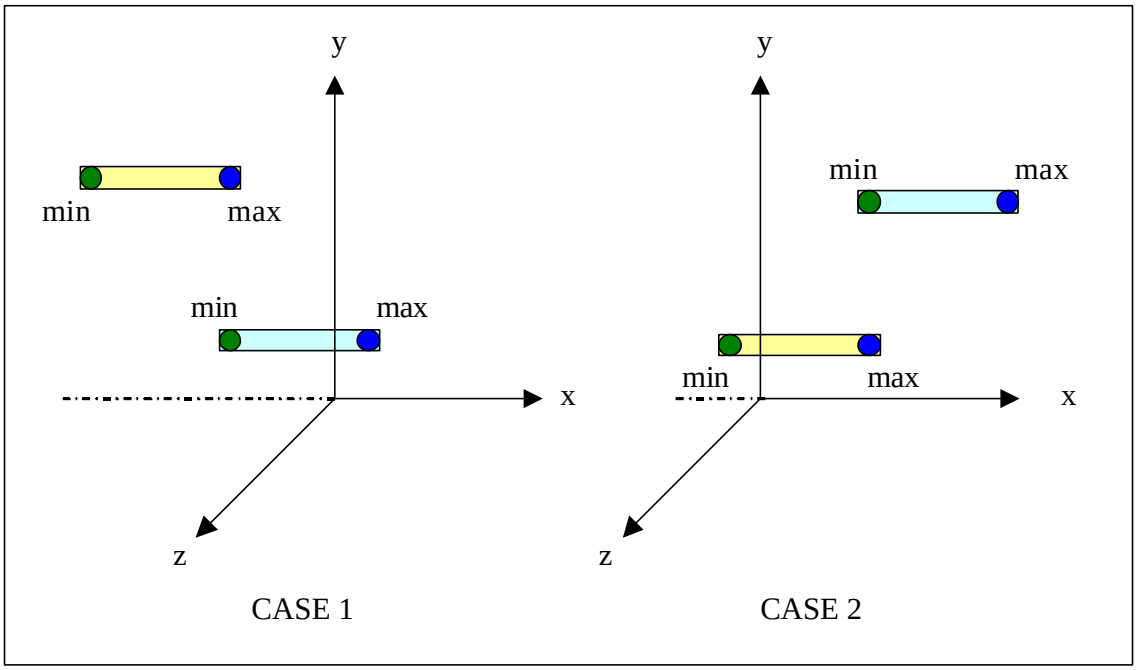


Figure 14. The Possible Locations on the X-Axis.

After determining the shifting vector for the geometry, the vector is added to all vertexes in the input file. Then the new values are saved into the output file. The shifting values need to be saved, too. After pasting the new vertex values back to their geometry nodes, the segments are translated to their proper location in the humanoid body by using the inverse shifting values. Multiplying the shifting values with minus one will give the inverse shifting value.

Inverse shifting values can be directly set to the translation field of the Joint node. But doing so breaks the nested-joint structure of the skeleton. The translations depend on their parents' translations. Rebuilding the new humanoid with the new vertexes requires one to follow the hierarchy in the nested-joint structure. For instance, the shoulder (upperarm) is translated before the elbow (forearm). Vectors v1, v2 and v3 in Figure 15 represent the inverse shifting vectors obtained above. Vectors v2 and v4 are the *shifting vectors* relative to the parent joints. It is assumed that v1 is applied to the highest-level parent joint and a first level child joint is connected to it. The actual location of the first-level child is v2 away from the parent joint, and v3 away from the origin. v1 and v3 are known (inverse shifting vectors), and v2 is equal to the subtraction of v1 from v3. That is, inverse shifting vector of the parent is subtracted from the child's inverse shifting vector. v2 is set to the translation field of the first-level child joint. The same principle is applied to the second-level child joint. This time v3 and v5 are known, but v4 is unknown. The humanoid Andy using the technique described above is shown in Figure 16. Humanoid Andy has the same geometry as humanoid AndyLow.

3. Getting Humanoid Andy to Work with Java Networking

A Browser or Plug-in is an application used to view VRML, X3D and/or other 3D files. Multiple browsers are available, supporting features in the various Web3d specifications. Cortona and Cosmo Player are two of the browsers listed on WEB3D web site [Ref.15]. Both Cortona and Cosmo Player are compatible with MS Internet Explorer and Netscape Navigator and support java scripting. MS Internet Explorer (IE) has an advantage over Netscape Navigator by virtue of its security policy. Because of obvious security reasons, applets running in a browser may not send network traffic to nor receive any hosts other than the server where applets are downloaded from without throwing a security exception. This problem might be encountered when the scripting class is connected to the network and receives the updated data of the Humanoid through the network. Internet Explorer gets around with this security rule by its security policy. According to the policy Java classes in a *.jar file, which are in the same directory on the local disk as the invoking web page, are not subject to applet security rules. For this reason, Internet Explorer is preferred in this thesis.

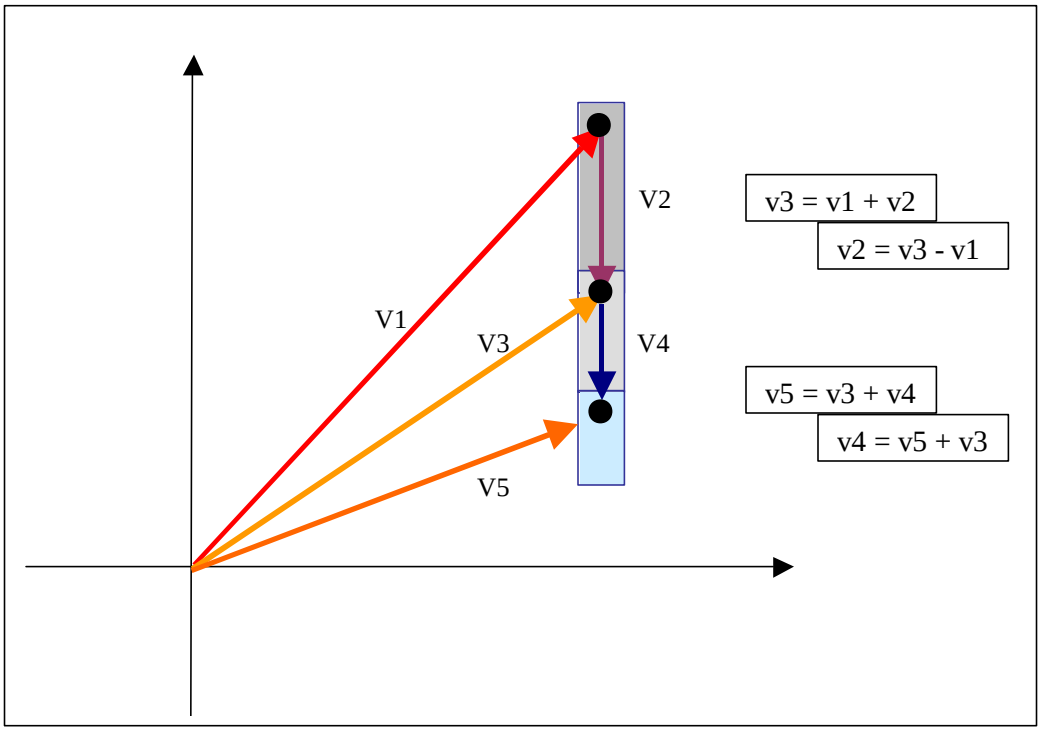


Figure 15. Rebuilding the New Humanoid in the Nested Joint Structure.

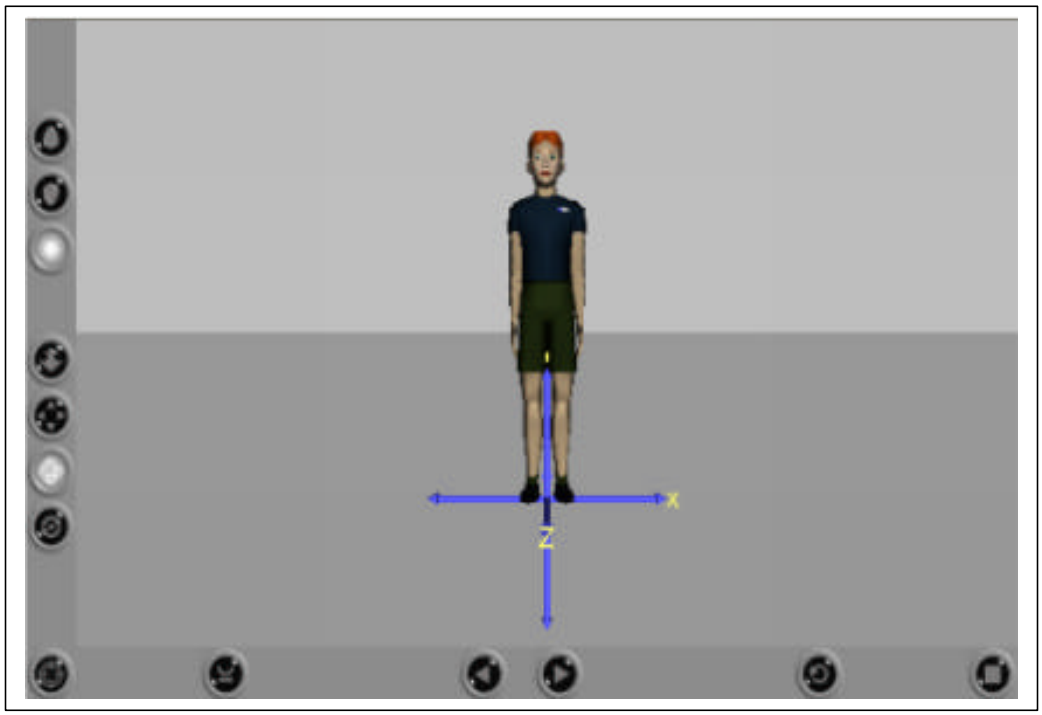


Figure 16. Humanoid Andy.

In order to implement IE's security policy, the following tasks need to be performed. First, put the compiled java classes into the same *.jar file in which the VRML or X3D file is placed. A batch file can be created for this purpose. The sample batch file lines below compile MyScript.java and add all the files and the directories located in the current local directory into MyJarFile.jar file.

C:\jdk1.3.1\bin\javac MyScript.java

*C:\jdk1.3.1\bin\jar cvf MyJarFile.jar *.**

Second, add the classpath for the *.jar file to the "environment variables." The classpath can be added to Windows 2000 or Windows XP by selecting start > control panel > system > advanced > environment variable. Then the following code is typed.

CLASSPATH c:\path of the jar file\MyJarFile.jar

Although implementing Internet Explorer's security policy solves the security problem in java networking, it is still not enough to get Java to work with X3D. Some nodes, such as Transform and HAnimJoint, in X3D provide fields that are capable of receiving and/or sending data through ROUTE nodes. Route nodes are forwarding nodes and contain four attributes: fromNode, fromField, toNode, and toField. The idea is simple: *from* source *to* destination. The source and the destination can be any node, such as Transform and Script node. The data of fromField is transferred to toField if these fields are defined properly. An EventIn event allows a field to receive data and an EventOut event to send data. The Rotation field defined in Transform node supports both the EventIn and EventOut event that makes it capable of receiving and sending data. Thus the fromField should support EventOut event, and the toField should support the EventIn event. The relation between Java and X3D is illustrated in Figure 17.

Script node is the key node for the link between Java and X3D. This node contains one or more fields that support either EventIn's or EventOut's or both. These fields are the pipes and need to be known by the Java script class referred inside the Script node. For this reason all these fields are defined and created in the Java class and the class is inherited from the Script class. Figures 18 and 19 show the relationship.

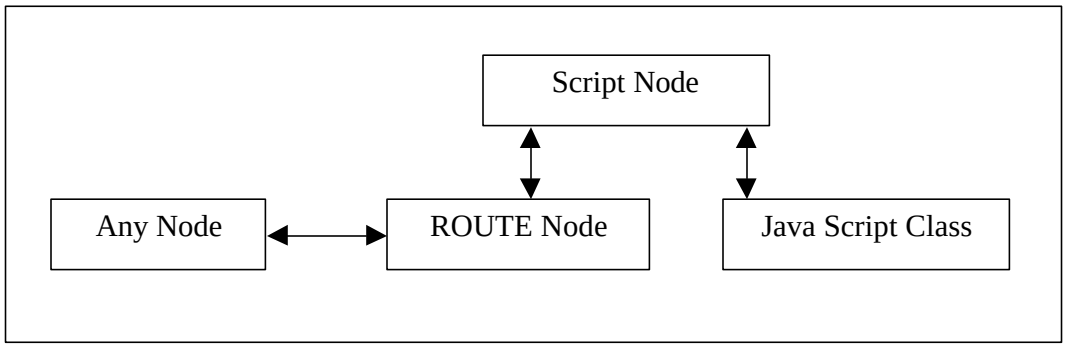


Figure 17. The Relationship between Java and X3D.

```
import vrml.*;

public class MyScript extends Script {
        // A reference to the fields in Script node (X3D)
        protected SFRotation leftHip;
        protected SFRotation rightWrist;

        // Initialize the Script node
        protected void initialize () {
                // Create connections
                leftHip = (SFRotation) getEventOut ("leftHip");
                rightWrist = (SFRotation) getEventOut ("rightWrist");

                // Setting an initial value
                leftHip.setValue (0.0f, 0.0f, 1.0f, 0.0f);
                rightWrist.setValue (0.0f, 0.0f, 1.0f, 0.0f);
        }
}
```

Figure 18. The Java Script Code.

4. Implementing Multiple MARG Sensors to Track Humanoid Andy

Humanoid Andy has a nested-joints structure as mentioned earlier. The HumanoidRoot is the root joint and the parent joint for the second order joints. Second-order joints are the left hip, right hip and vl5. For example, the hierarchical joint structure for the right arm is as follows: HumanoidRoot, vl5, r_shoulder, r_elbow, r_wrist and r_fingers. A MARG sensor is capable of providing quaternion data for the motion of the attached segment. Positioning the tracked segment is possible through implementing the quaternion data to the related joint. After converting the quaternion data into an axis-

angle pair data, the SFRotation field of the Joint node sets the axis-angle data pair. This process is enough for only one sensor tracking. Additional processes are required to implement multiple sensors to the humanoid Andy.

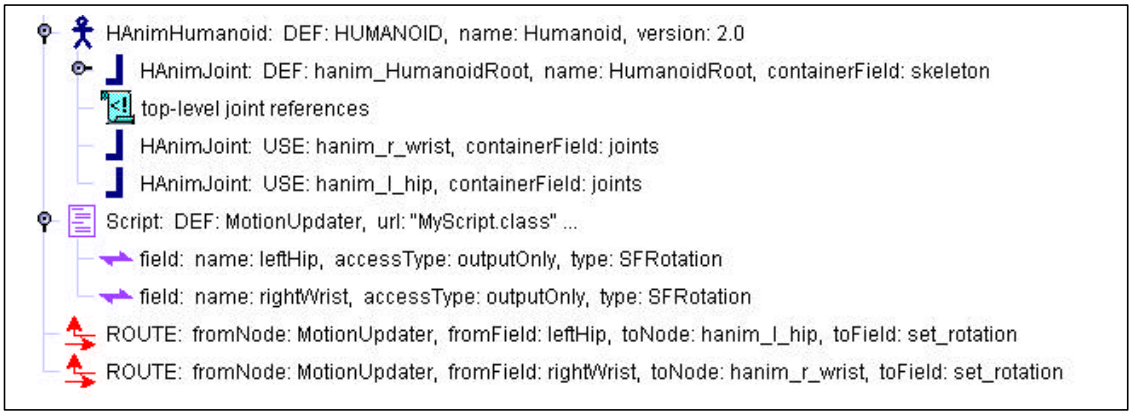


Figure 19. The X3D Script Node.

Let's assume a simple tracking system with one MARG sensor. If the quaternion output of the MARG sensor is q, then the total orientation data is only q (see Figure 20). But, when tracking a nested structure with two segments, each MARG sensor will produce quaternion data (t, h) for its own measurement, which is completely independent of the other. Implementing t to the first joint will not only position the first segment accurately but also affects the second joint, so that the starting position for the second segment is set to t. This will result in an inaccurate positioning for the second segment if h is directly implemented to its relevant joint (see Figure 21). In order to solve this problem, initially the inverse quaternion of t is implemented to the second joint to eliminate the effect of first MARG sensor. Later, quaternion h can be safely implemented to the second joint. The formulation of this process is as follows. Assume

h = quaternion for child joint motion measured by the MARG sensor,

t = quaternion for parent joint motion measured by the MARG sensor,

c = final quaternion for the child joint.

The final quaternion data (c) applied to the joint is calculated as follows:

c = inverse (t) * h.

The idea for positioning the nested structures with more than two segments is identical for those with two segments. The inverse motion for the parent joint(s) is multiplied with the child joints motion. This multiplication is carried out in quaternion type.

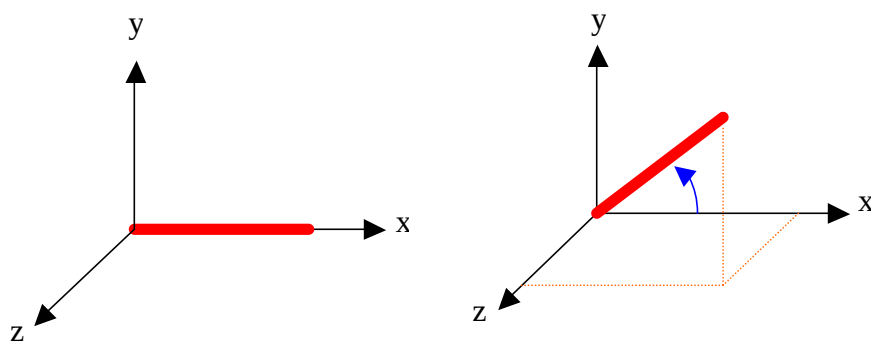


Figure 20. Positioning One Segment.

D. SUMMARY

This chapter first presented the H-Anim standards and the X3D graphics language. It continued with providing information about H-Anim nodes contained by X3D and the nested-joint structure of humanoids. Finally, the methods used for rebuilding the humanoid Andy and getting humanoid Andy work with MARG sensors in the network environment was discussed. For this thesis, a low-resolution Humanoid with fifteen joints is sufficient, but a high resolution Humanoid is needed for fine gestures and mimics in further work.

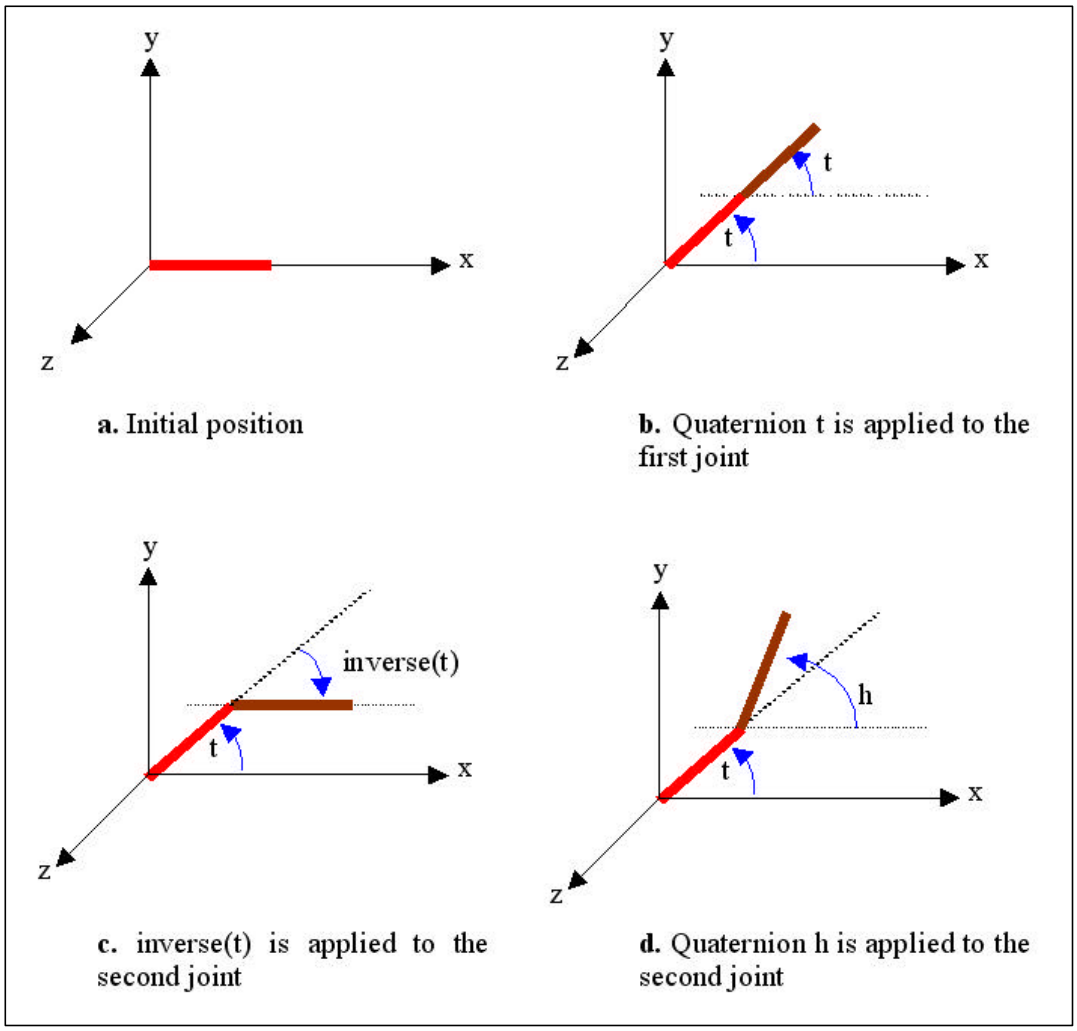


Figure 21. Positioning Two Adjacent Segments in the Nested Structure.

IV. DESIGN OF THE CLIENT-SERVER PROGRAM

Multicasting is the most efficient way of transmitting information among a large number of group members spread out over different networks. Reduction of network bandwidth use is the major advantage of using multicasting protocols. Unfortunately, most of the routers on the Internet do not have multicast routing protocols configured. A technique called *tunneling* is used to overcome this problem. Because very few numbers of clients, up to 10 or 20, are required in this thesis, an alternative method to the multicasting protocol referred as *Multicasting Using TCP and UDP Protocol (MUTUP)* is introduced. This method overcomes the tunneling problem by implementing a multicasting protocol with TCP and UDP protocols.

Implementing a multicasting protocol using TCP and UDP protocols requires a concurrent client-server structure. Concurrent processing in Java can be implemented with Thread classes. Each thread employed in the system requires additional memory and CPU usage. The increase in the number of the clients supported by the server program will increase the amount of memory consumed. This might result in out-of-memory problems or very low performance due to the overloaded CPU. In order to provide better performance and avoid memory problems, the user in server settings restricts the number of clients.

In principle, MUTUP is very similar to the File Transport Protocol (FTP). The main difference between them is the protocols that are used to establish connections between the server and the client applications. Two connections are set for handling a client in both methods. These two connections use TCP protocol in FTP protocol, whereas one is TCP and the other is UDP in MUTUP. The reason for this difference is that file transfer requires reliable communication, but motion data streaming requires fast communication.

This chapter discusses MUTUP developed as an alternative for multicasting protocol used in this thesis. It also discusses the implementation of this MUTUP to the MARG project.

A. MULTICASTING USING TCP AND UDP PROTOCOLS

MUTUP sets a TCP and a UDP connection between the server and each of the client applications. A TCP connection is established as follows: First, the Server class creates a ServerSocket object and waits for clients to make a request to set a TCP connection. In order to make a request, clients need to know the IP address and the TCP port number of the server. This information can be either assigned as a fixed global variable or provided externally in the command line when running the client application. Second, the client creates a Socket object with the provided or assigned IP address and TCP port number. The Socket object makes a request of a TCP connection from the ServerSocket object in the Server class. The ServerSocket object accepts the request and sets a TCP connection by creating a Socket object for the requesting client. Third, since there will be multiple clients in the system, the Server class creates and starts a Thread object for handling each new client. A class extending from the Runnable interface is required as a parameter to the Thread class. Therefore, an additional class referred as HandleClient class is defined for handling clients. A pseudo code for the Server class and the HandleClient class is provided in Figures 22 and 23.

A TCP connection by itself is not enough to create an alternative design to multicasting for this thesis. A TCP connection is used for general-purpose communications, such as transferring essential data for setting a UDP connection and informing the server when the client wants to terminate. Since the motion data tracked by the MARG sensor(s) require low latency, the second link between the client and the server must be a UDP connection.

The number of clients accepted by the Server class is limited by using a counter variable in an infinite loop. Since the number of active clients at a time is dynamic, the counter variable is increased after each client's request and decreased after each client shuts down. If the counter reaches the upper allowable limit, then it stops accepting new clients until one of the active clients shuts the connection down.

```
class Server  {
    Set counter = 0;                                    // the counter
    Set maxClientAllowed,                               // Maximum clients allowed

    Create the shared array/memory,                     // byte[] sharedMemory

    Create and start the thread for MemoryUpdater class

    create ServerSocket object;

    while (true) {
        if (counter < maxClientAllowed) {
            Accept new client;                          // ServerSocket.accept()
            Create Socket object for the client;
            Create Thread object to handle the client;  // new Thread(new HandleClient(sock, this))
            Start thread;                               // Thread.start()
            counter ++;
        }
    }
}

Provide the method for decreasing the counter,
Provide the setter and getter methods for the shared array.
```

Figure 22. Pseudo Code for the Server Class.

The constructor of the HandleClient class accepts the Socket object passed from the Server class, and extracts the input and output streams from the Socket object. These streams are used for the TCP communication. The second connection is established in the HandleClient class by requesting the port number and the IP address of the UDP connection from the Client class. The request is done through the TCP connection. The Client class creates a DatagramSocket object after receiving the request from the HandleClient class and responds after retrieving the UDP port number and IP address that the DatagramSocket object listens for. The HandleClient class sends feedback to the Client class to confirm that the requested data is received. A loop is employed in the HandleClient class in order to create DatagramPacket objects (packet) for the data. The IP address and the UDP port number of the client are set to each packet before sending them. The Client class employs a loop for receiving the packets. The loop continues until no more data is left or the UDP connection is broken. Finally, the TCP and the UDP

connections are killed on both sides and the counter variable of the Server class is decremented. The pseudo code for the Client class is provided in Figure 24.

```
class HandleClient implements Runnable

CONSTRUCTOR:
Save the Socket object and the Server object passed by the Sever class;
        // Socket sockTCP = Socket Object passed by the server;
        // Server owner   = this;

RUN:
Create input/output stream for the TCP connection;
        // DataInputStream dis = new DataInputstream (sockTCP.getInputStream ());
        // DataOutputStream dos = new DataOutputstream (sockTCP.getOutputStream ());

Request address and port number for the UDP connection
        // dos.writeUTF("Requesting address and port number for UDP connection");
        // dos.flush ();

Wait to receive the respond from the client
        // clientRespond = dis.readUTF();

 Send acknowledge to the client;
        // dos.writeUTF("address and port number received");
        // dos.flush();

 Tokenize address and port number from the respond message
        // StringTokenizer st  = new StringTokenizer(clientMsg);
        // InetAddress destAddrUDP = InetAddress.getByName(st.nextToken());
        // int  destPortUDP = new Integer(st.nextToken()).intValue();

 while (more data to send) {
        create DatagramSocket
                // sockUDP  = new DatagramSocket();  -- bounds to any available local port --

        read the motion data from the shared byte array in the server
                // byte[] barray = owner.getSharedByteArray();

        create a DatagramPacket object,
        set the address and the UDP port number of client
                // packet = new DatagramPacket(barray, barray.length, destAddrUDP, destPortUDP);

        send the packet
                // sockUDP.send(packet);
}

Kill the UDP and the TCP connections,
Decrease the counter variable of the Server class      // owner.decreaseCounter();
```

Figure 23. Pseudo Code for the HandleClient Class.

The method described above handles multiple clients concurrently, but it is not explained how the same motion data is distributed to the clients simultaneously. Unless the same data is distributed to the clients simultaneously, the system will not implement multicasting. As a solution, a shared array/memory for the motion data is used in the system. In order to get the shared array accessible by all client handler threads, the array is defined in a global perspective, i.e. in the Server class, and the getter and setter methods are provided. For updating the shared array, the Server class employs an additional thread implementing the MemoryUpdater class. The pseudo code for MemoryUpdater class is provided in Figure 25.

```
class Client

        Create Socket object for TCP connection
                // addrTCP = InetAddress.getByName(ADDRESS)
                // sockTCP = new Socket(addrTCP, DEFAULT_TCP_PORT);

        Create input/output stream for the TCP connection;
                // DataInputStream  dis = new DataInputstream(sockTCP.getInputStream());
                // DataOutputStream dos = new DataOutputstream(sockTCP.getOutputStream());

        Wait to receive request from the server
                // serverRequest = dis.readUTF()

        Create DatagramSocket object for UDP connection
                // sockUDP = new DatagramSocket();

        Retrieve IP address and UDP port number from the DatagramSocket object
                // addrUDP = (InetAddress.getLocalHost()).getHostAddress();
                // portUDP = sockUDP.getLocalPort();

        Respond to the server's request
                // dos.writeUTF(addrUDP+" "+portUDP");
                // dos.flush();

        Wait for acknowledge
                // dis.readUTF();

        while (not finished) {
                Receive motion data
                        // sockUDP.receive (packet);

                Update Humanoid
        }

        Kill the TCP and UDP connections,
```

Figure 24. Pseudo Code for the Client Class.

```
class MemoryUpdater implements Runnable

CONSTRUCTOR:
save the Server object passed by the Sever class;
        // Server owner   = this;

RUN:
Create 3 arrays for raw sensor data,

Create and run a thread implementing MARGDataUpdater class,
// This thread receives data through 3 MARGDataReader class,
// and updates raw sensor data arrays continuously

while (continue reading data from sensors) {
        Convert the 3 raw sensor data into quaternion data,
        // Use QuestQuaternionProducer class

        Convert the quaternion data into byte[] type data,

        Save the converted data into the shared array,
        // by using setter method provided by the Server class
        // owner.setSharedByteArray(barray);
}
```

Figure 25. Pseudo Code for the MemoryUpdater Class.

B. IMPLEMENTATION OF MUTUP IN THE MARG PROJECT

The Concurrent Client Server Program is used as a connector between the MARG sensor(s) and the representation of the humanoid developed in X3D to animate the tracked human motion. The humanoid used in this design is the humanoid Andy introduced in Chapter III. Humanoid Andy is capable of animating the 15 MARG sensor data that is the minimum requirement for full-body human motion tracking as mentioned in [Ref. 2]. The block diagram of the Concurrent Client Server Program implementing MUTUP is illustrated in Figures 26 and 27.

The StartServer class is the starting program for the Concurrent Client-Server Application (CCSA) and triggers the server program by creating and starting a Thread object for the Server class. The Server class implements Runnable interface and is responsible for receiving raw MARG sensor(s) data, converting them into quaternion representation and updating the shared array/memory continuously. Furthermore, the Server class is responsible for starting a new thread for handling each new client and controlling the number of active clients at any given time. Since the shared array for the

motion data is required to be accessible by all sub-threads in the system for implementing multicasting, the Server class contains the shared array. In this way, the shared array gains a global status over the threads created by the Server class and both the thread implementing MemoryUpdater class and threads implementing HandleClient classes can access the shared array. The Server class provides setter and getter methods for the shared array. For the purpose of reducing computation process, the shared array is defined as byte[] type. This is because the motion data is transmitted to the clients through UDP connections and UDP packets accept payload data in byte[] type. Otherwise, each HandleClient class has to repeat the same conversion process from high-level quaternion data to the low-level byte[] data. The number of threads employed by the Server class depends on the number of clients allowed to be handled. In any case, a thread for implementing MemoryUpdater class is created and started first because the server needs to be ready for capturing motion data before accepting any client. Once the MemoryUpdater class is created and started to update the shared array, the clients can be accepted within the allowed limitations.

The MemoryUpdater class implements the Runnable interface and its responsibilities are as follows: Receiving the sensor data through three different WiSER2400.IP wireless serial adapters and converting the raw sensor data first into quaternion representations and later into byte[] form. Receiving the sensor data through the 3-channel CIU is more challenging than through the 15-channel CIU. Since the 15-channel CIU is not ready at this moment, the MemoryUpdater class is designed to use the 3-channel CIU connected to three different WiSER2400.IP adapters and MARG sensors. The MemoryUpdater class will be much simpler for a 15-channel CIU since it will use only one wireless serial adapter and all sensors will be connected to this serial adapter. In order to use the 3-channel CIU in the concurrent client-server design, an additional thread implementing the MARGDataUpdater class is needed. The MARGDataUpdater class is in charge of three different MARGDataReader class for handling the three MARG sensors connected to the 3-channel CIU. The MARGDataReader class is responsible only for receiving the sensor data through the WiSER2400.IP serial interface and parsing it. Handling these parsed raw sensor data is under MARGDataUpdater class's responsibility. The MemoryUpdater class provides three different arrays for each parsed

raw sensor data. These arrays are in int[] type and updated simultaneously by the MARGDataUpdater class.

The second task of the MemoryUpdater class is to convert the raw sensor data into usable quaternion data. The QuestQuaternionProducer class implements the Quest quaternion algorithm introduced in [Ref. 7] in order to handle this conversion process. The MemoryUpdater class uses three different QuestQuaternionProducer classes for each raw sensor data since each sensor calibration data is different from the others and the Quest quaternion algorithm uses the sensor calibration data in its calculations. The output quaternion data is in [w x y z] order. Each element of quaternion data is in double type data. It is acceptable to save the quaternion data as it is, but an additional conversion into byte array form is needed before sending them to the clients. When multiple clients accepted by the server, this means redundant computation overload on the computer running the server application. Therefore, this conversion task is the third responsibility of the MemoryUpdater class. The size of the shared array is flexible for further work with 15 sensors, or even more than 15. That is, it is capable of handling 15 sensor data in byte-array from. For this reason, the MemoryUpdater class can update any selected three-joint location in the shared array by setting the number of joints to the shared array updater method. The three joints are determined globally either in a different class or within the Server class. Using a different class is more flexible than defining within the Server class.

As described earlier in the chapter, the HandleClient class is used to handle each visiting client. The HandleClient class implements Runnable interface and establishes a UDP connection with the client, which it is responsible to handle. Since setter and getter methods are provided for the shared array by the Server class, the HandleClient class reads the latest updated MARG sensors data by using the getter method. The read data is transmitted to the client without any additional calculations: simply read and transmit.

On the client side, the humanoid Andy is connected to the ClientReceiver (Client) class by using a Script node and can receive the MARG sensors data through this class. ClientReceiver class sets two connections with the server program. First, a TCP connection for general-purpose communications is set. The client conducts the initial contact with the ServerSocket object of the Server class by using the TCP connection.

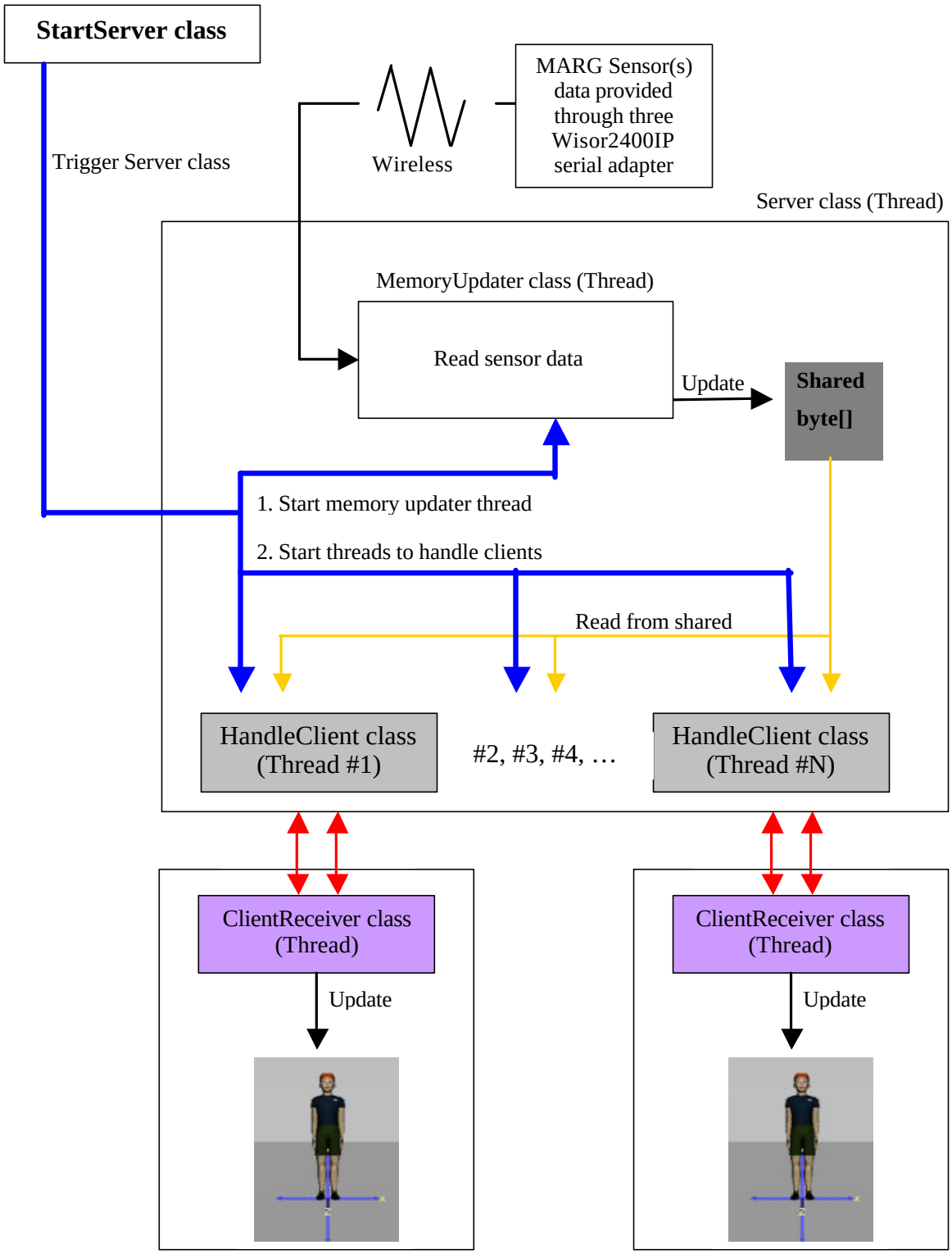


Figure 26. Block Diagram of Concurrent Client Server Program.

If the ServerSocket object accepts the client, then the second connection is established between the ClientReceiver and the HandleClient class for transferring the MARG sensors data. Since the data rate is the major issue regarding performance in networked virtual environments, the second connection is implemented using the UDP/IP protocol.

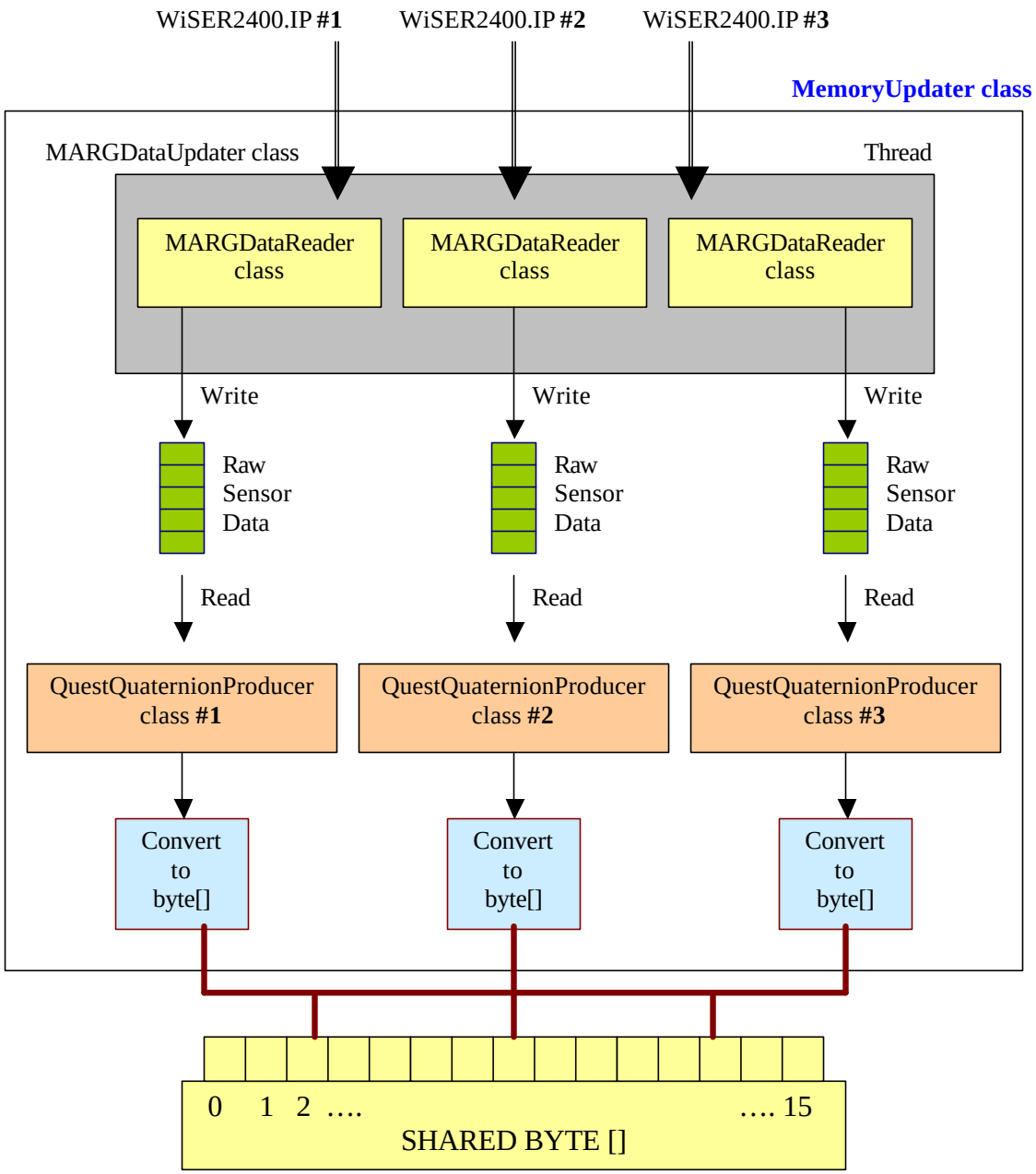


Figure 27. A Detailed Block Diagram of MemoryUpdater Class.

C. SUMMARY

This chapter discussed MUTUP developed as an alternative of the multicasting protocol to be used in this thesis and discussed the implementation of MUTUP to the MARG project. MUTUP overcomes the tunneling limitations/requirements of the multicasting protocol. The major drawback of MUTUP in a concurrent clients-server application is the limited number of clients accepted by the server. But, this drawback does not affect the MARG project since a limited number of clients is expected. The principle of MUTUP is similar to the FTP protocol, except the types of connections established between the client and the server applications.

THIS PAGE INTENTIONALLY LEFT BLANK

V. TESTING AND EVALUATION

This chapter describes the efforts of testing and developing procedures of the humanoid Andy and the concurrent client-server program. Tests are conducted with simulation data produced by a program developed in the Advanced Physically Based Modeling course [Ref. 16] and real-time data obtained by using the MARG sensors.

A. ROTATIONAL MOTION TESTS THE HUMANOID ANDY

As described in Chapter III, humanoid AndyLow was created in VRML97 and Proto nodes were declared to implement the H-Anim standards. Humanoid Andy, the modified version of the humanoid AndyLow, is compatible with X3D and implements H-Anim standards using built-in humanoid nodes provided by X3D. The process of developing the humanoid Andy from the humanoid AndyLow was discussed in Chapter III. Orientation tests applied to the joints of the humanoid Andy are required for validating the humanoid. The first orientation test is implemented using the OrientationInterpolator node provided by X3D. The OrientationInterpolator node generates a series of rotation values in axis-angle pair data type that can be routed to the Joint nodes of the humanoid Andy. This test showed that the humanoid Andy could be used for tracking human motion since all 15 segments rotated about their relevant joints (connection points to the parents). Figure 28 illustrates the simultaneous rotation of the left forearm about z-axis and left hand about y-axis. Figure 29 illustrates the rotations of the left forearm, right forearm and the head of humanoid Andy.

The second orientation test is conducted using simulated data created by a program developed in the Advanced Physically Based Modeling course [Ref. 16]. Methods in LISP codes provided by the instructor were used in developing the program. The program produces three-degrees-of-freedom (3 DOF) motion data for a rigid-body model of a human arm either in Euler angles or in quaternions. The resulting data is displayed on the screen and is saved into a file either as Euler angles or as quaternion data. The mouse and a button on the keyboard are used to move the arm in 3 DOF. A sample screen capture of the LISP program moving an arm in 3 DOF is illustrated in

Figure 30. Subsequently, the quaternion data saved in the file is read by a Java program and set to the Joints of the humanoid Andy. Since the humanoid Andy cannot implement quaternion data directly for rotations, a computation from quaternion into axis-angle pair is processed before setting the Joint nodes.

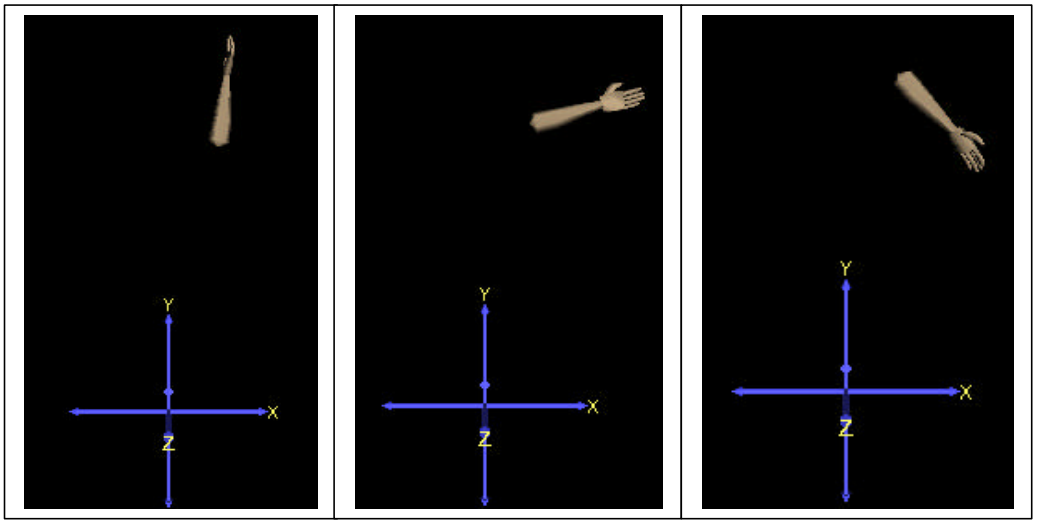


Figure 28. Simultaneous Rotation of the Left Forearm about Z-Axis and Left Hand about Y-Axis

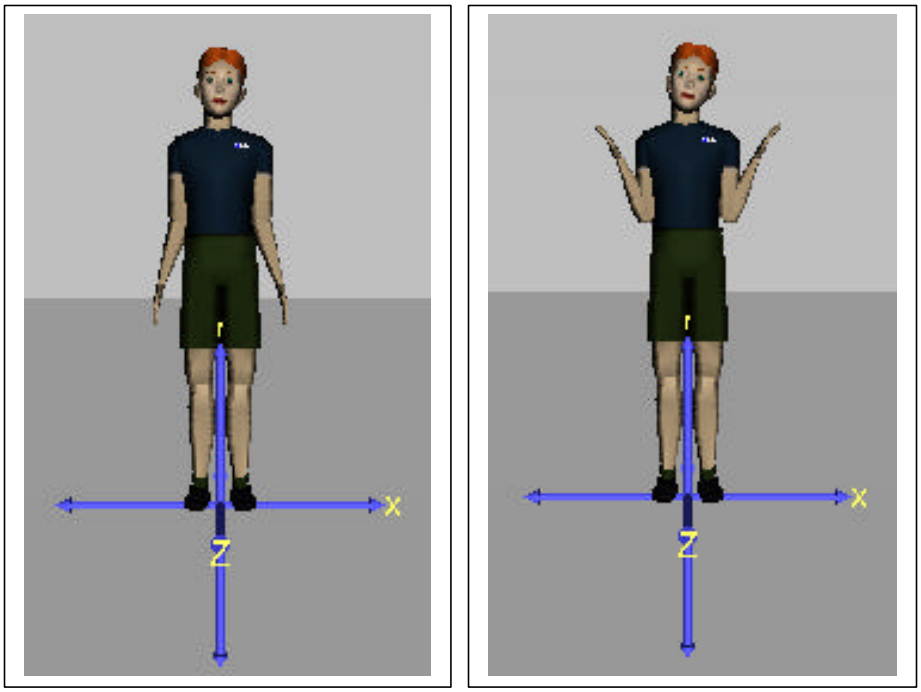


Figure 29. Rotations of Left Forearm, Right Forearm and Head of the Humanoid Andy

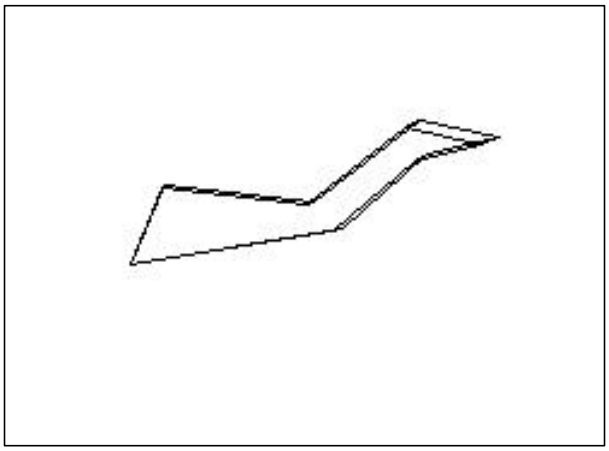


Figure 30. A Sample Screen Capture of the LISP Program Moving an Arm in 3 DOF

B. TESTING HUMANOID ANDY WITH ONE MARG SENSOR

Even though the orientation tests discussed above are not in real time, they showed that humanoid Andy is ready for human motion tracking using MARG sensors in a real-time application. The testing with the MARG sensor follows three steps. First, the MARG sensors capture limb segment independently from the other sensors and cannot be directly implemented to the joints. Therefore the humanoid is tested for one MARG sensor at the beginning using one-channel CIU. Second, in order to achieve a real-time application a server and a client program are developed. The server is responsible only for reading motion data of one MARG sensor, and then transmitting it to the client program. Humanoid Andy receives the real-time motion data through the client program. A UDP connection between the server and the client program is set for achieving a higher data rate.

Third, the data read from the sensor is raw data and needed to be converted to a high-level representation. A joint work with [Ref. 8] is started to implement this conversion. The Quest and Triad algorithms introduced in [Ref. 7] can be used to determine the three-axis attitude from vector observations. Both algorithms provide low-cost computation and high numerical accuracy. Since the Quest algorithm is chosen for higher accuracy applications, this thesis focused on implementing the Quest algorithm. The Quest algorithm uses six accelerometer and magnetometer data read from the sensor as an input and produces quaternion data for the motion. Figure 31 shows the high-

accuracy of rotations obtained by the MARG sensors using the Quest algorithm. The arm is initially parallel to the y-axis facing down to the negative y-axis. It went through a rotation of 90 degrees about negative x-axis.

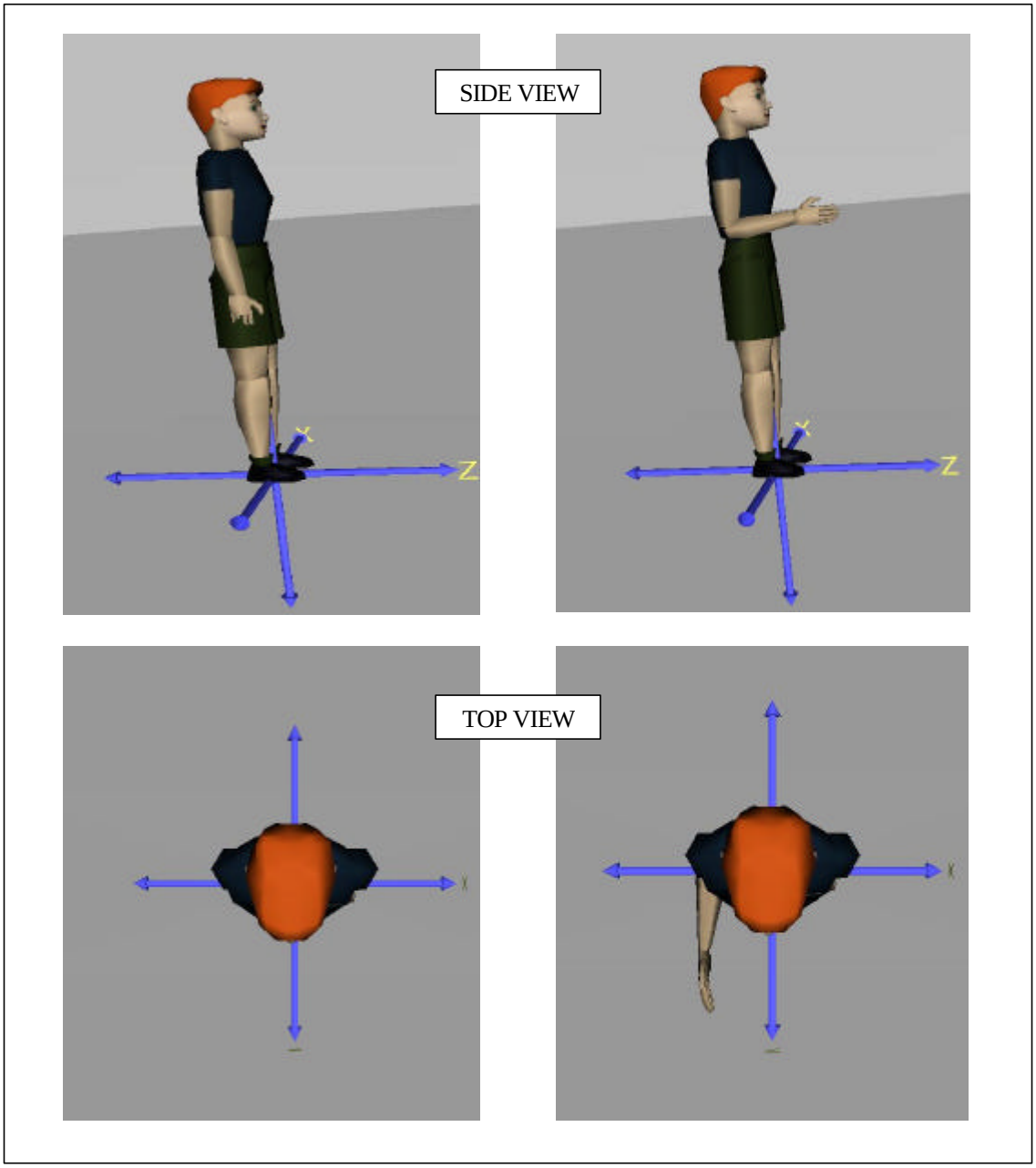


Figure 31. The Implementation of Quest Algorithm to the Humanoid Andy (90 Degrees of Rotation about Negative X-Axis)

C. TESTING HUMANOID ANDY USING TWO MARG SENSORS

Use of the quaternion data from multiple MARG sensors in a nested-joint structure is another testing applied to the humanoid Andy. Each MARG sensor works independently from each other. But, the joints in the skeleton of the humanoid are dependent on the adjacent parent joints. As described in Chapter III, the inverse motion of the parent segment is multiplied with the motion of the current segment before applying it to the relevant joint of the current segment. The concurrent client-server program discussed in Chapter IV is used to apply this method for testing. Two adjacent segments selected for the testing are the right upperarm and the right forearm. The related joints for these segments that are in the connection point to their parent joints are the *r_shoulder* and *r_elbow* joints. Two MARG sensors are connected to the three-channel CIU and their calibration results are set to the corresponding QuestQuaternionProducer classes. Each MARG sensor needs to be calibrated independently from the other sensors. In this case, two different objects of the QuestQuaternionProducer class are created in the MemoryUpdater class. In addition to these objects, two different MARGDataReader objects are also created for reading sensor data from two different MARG sensors. Finally, the system is set to support two MARG sensors on the server side. The quaternion data for each sensor is produced independently of the others, each using its own calibration data. The produced quaternion data is converted into byte[] type data and saved to the corresponding location in the shared array. The HandleClient class reads the shared array and transmits it to the client program that runs the humanoid Andy. At this moment, the quaternion data transmitted to the client side is the exact representation of each MARG sensor measurement.

On the client side there are three array-type data members defined for storing the motion data received from the server program. The reason for defining three different data members for holding the motion data is to provide flexibility to the further client programs with different humanoids. The humanoids developed in the future may use one of these data members, depending on their structure. The first data member stores the exact sensor measurements in high-level (double) quaternion type data. The client program converts the low-level byte[] type data received in the packet to a high-level quaternion type data and saves it into the first data member. The second data member

stores the motion data in high-level quaternion data but ready to apply to the nested-joint structure of the humanoid Andy. To get the sensor data compatible with humanoid Andy, the method of taking the inverse motion of the parent joint and multiplying it with the original measurement is applied, as described in Chapter III. For this reason, the original sensor data stored in the first data member is used for these calculations. Finally, the third data member is defined in order to get the sensor data to work with X3D. Since X3D accepts axis-angle type data instead of quaternion type data, the motion data saved in the second data member is converted into the axis-angle pair representations and saved in the third data member. The *Quat4d* and *AxisAngle4d* classes provided in the *javax.vecmath* package handle conversions from quaternion into axis-angle pair. This package is an open-source code included in the *Java3d1.3.1* package [Ref. 17].

After obtaining the axis-angle pair data, the joints of the humanoid Andy is set to the third data member values. The result of using two MARG sensors in the system is illustrated in Figures 32 and 33.

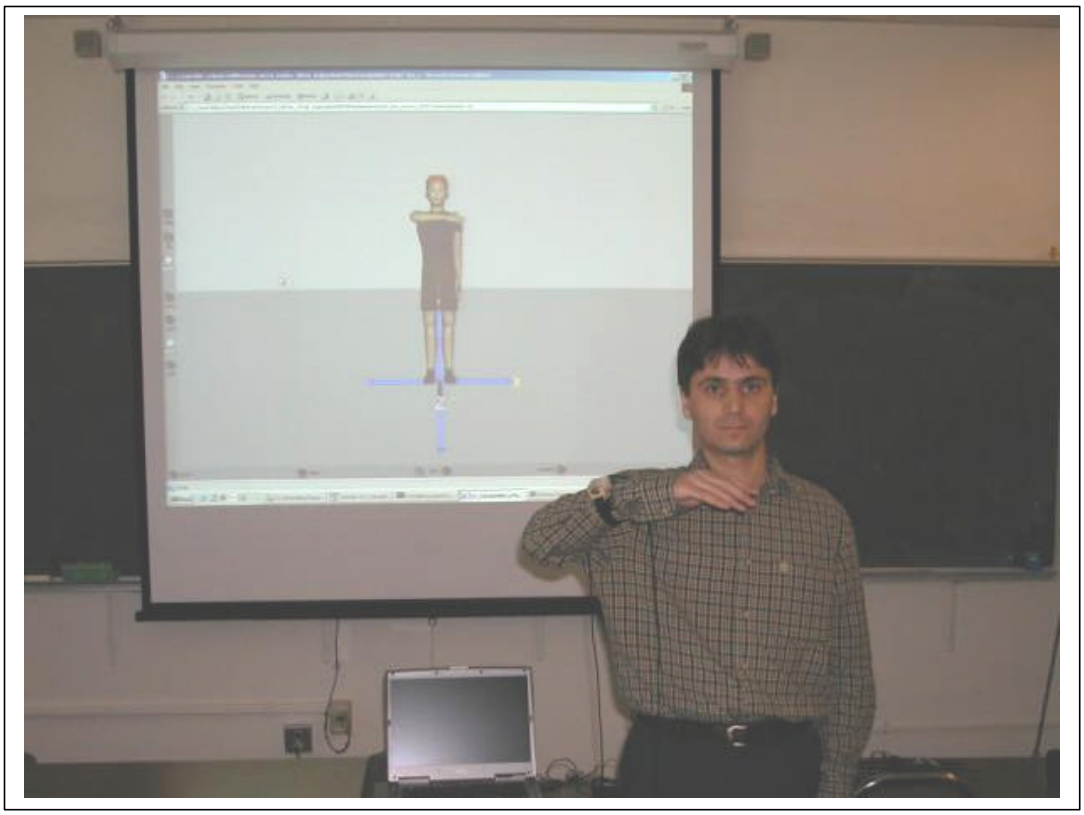


Figure 32. Testing Two MARG Sensors on the Humanoid Andy.

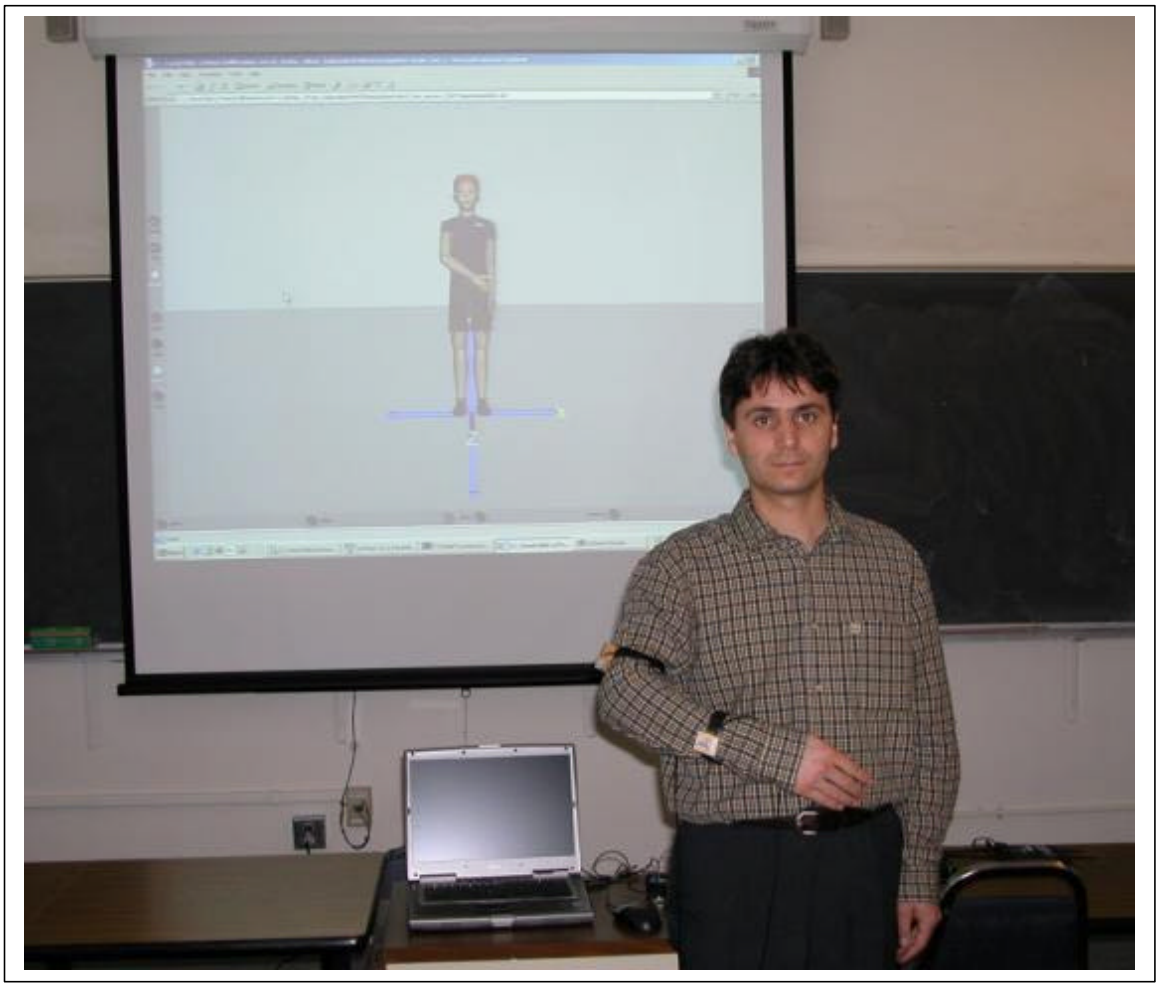


Figure 33. Testing Two MARG Sensors on the Humanoid Andy.

D. TESTING THE CONCURRENT CLIENT-SERVER PROGRAM

The concurrent client-server program was first tested using random data. The double type random data are converted to byte[] type data and then transmitted to the clients. The client programs in this testing do not run humanoids and the MARG sensors are not connected to the system. Both the server program and the client programs are all running on the same computer. The highest number of clients accepted by the server program is not limited, but the system is only tested with up to five clients running at the same time. The system is also tested on the LAN and on the Internet successfully. Three computers each running a client program and a computer running both the server program and a client program are connected to the network. The clients did not join to the

system at the same time, but the same data was delivered concurrently to the clients successfully. Even if a client program quits the system and rejoins to the system later, all active clients receive the same data simultaneously. That is, the system is multicasting. This property of the client-server program makes it possible for any new client to join and quit at any time. The only restriction is the number of clients allowed to be accepted by the server program. The user can control this number.

Another testing is applied to the system with three MARG sensors capturing human limb motion simultaneously. The client program runs the humanoid Andy. A client and a server program run on the same computer. The captured motion data is saved in the shared array on the server program. The size of the shared array is set to be large enough to hold 15-sensor data. Therefore, only the locations for the connected joints in the shared array are updated with the captured data and all the remaining locations retain the same value as they are initialized. In this way, the data transferred to the clients are in the size of 15 sensor data. This means, the system is greatly overloaded, but the goal of the MARG project is to get at least 15 MARG sensor capturing human limb motion simultaneously. The result of this testing is a successful animation on the client with almost no delay. A further step with several clients running on the Internet is not tested.

The performance of the WiSER2400.IP serial adapter in the system is not tested for 15 MARG sensors' data, but works well for one sensor. As described earlier, three different MARG sensors are connected to a three-channel CIU and each is handled independently by the server program. The server program reads the three sensor data simultaneously, as if the three sensor data are transmitted through one wireless serial adapter. The goal with the 15-channel CIU is to receive all 15 sensor data through one wireless serial adapter.

E. FINAL RESULTS

All the tests conducted above helped develop the humanoid Andy to animate 15 MARG sensor motions captured simultaneously and helped develop the humanoid Andy to work with the concurrent client-server program. Developing the concurrent client-server adds the MARG project the capability of capturing real-time human motion. The

overall system is not tested with 15-channel CIU at this moment. Minor modification to the system will make the system compatible with 15-channel CIU.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSIONS AND FUTURE WORK

This chapter presents the conclusions and the future work of this thesis, and briefly discusses what has been done and the lessons learned.

A. SUMMARY AND CONCLUSIONS

The existing MARG project prior to this research lacked a humanoid that met the need of animating 15 MARG sensor data. The existing humanoids developed for the project had their own limitations. One was very far from reality and did not follow the H-Anim standards. The other two humanoids were created using laser-scanned data and followed the H-Anim standards, but one had its adjacent joints broken and the other was capable of rotating only one joint. Therefore, the humanoid Andy was developed to meet the needs for animating the motion of a human measured by 15 MARG sensors.

A cartoon type humanoid AndyLow was selected as the starting point for developing the humanoid Andy. The humanoid AndyLow was developed using Proto nodes of VRML97 and implemented with H-Anim standards. X3D has the advantage of providing built-in humanoid nodes implementing H-Anim standards. First, the humanoid AndyLow was imported to the X3D and Proto nodes were replaced with built-in humanoid nodes. Although this process made the humanoid compatible with X3D, it was not possible to apply rotation to the segments about their connection points to the parent joints. An additional process discussed in Chapter III was applied to overcome this problem. Finally, the new humanoid was ready for animating 15 MARG sensor data and named as the humanoid Andy.

A network interface was missing in the MARG project. Adding networking capability to the project is vital to get a flexible system with real-time data streaming. The concurrent client-server application implementing multicasting using both the TCP and UDP protocols was developed. A shared array keeping the last update of the sensors' data was defined in the server program for a simultaneous data transmission between the server and the clients. Three WiSER2400.IP wireless serial adapters were connected to the three different MARG sensors through the 3-channel CIU. The server program listens

to the sensors' data through the UDP connections established wirelessly between the serial adapters and the server program. The humanoid Andy simulates the motion captured by the sensors' on the client side.

The major advantage of implementing multicasting using TCP and UDP protocols (MUTUP) with a shared array is overcoming the tunneling problem encountered in multicasting protocol. Today, most of the routers on the Internet cannot handle multicast packets and multicast packets are forwarded using unicast protocols. Using unicast protocol is referred as *tunneling*. The major disadvantage is the limitation on the number of clients handled by the server program at any time. There is a limitation because each client means an additional overload to the CPU and additional memory consumption that results in low performance or out-of-memory problems. Another drawback is the bandwidth restrictions. Sending motion data to any additional client adds additional traffic to the network. Despite these drawbacks, the MARG project is not affected, for a limited number of clients are needed. From this aspect, using MUTUP is advantageous over using multicasting protocol.

MUTUP has a similarity with the file transport protocol (FTP) in principle. The only difference is that the FTP sets two TCP links between the server and a client where MUTUP sets a TCP link for general-purpose communication and a UDP link for motion data transfer. Using TCP protocol for data streaming reduces the data rate.

The raw MARG sensor data consist of an accelerometer, a magnetometer and an angular rate sensor measurement. These measurements are converted to a quaternion data by implementing the Quest algorithm. Using quaternions in animation has a low-cost computation and high numerical accuracy [Ref. 7]. The quaternion data is produced separately for each sensor. Calibration data for each sensor differs from the others and is set to the Quest algorithm before producing the quaternions for the measurements. Each sensor mounted on the human body works independently of the others. Therefore a nested-joint structure cannot use the quaternions directly. An adjustment between the sensors' data is required to be conducted and to be compatible with the nested-joint structure of the skeleton of the humanoid Andy. The adjustment method is explained in detail in Chapter III.

In case future work needs a different humanoid with a different skeleton structure, the original sensor data is saved in a data member on the client program. The adjusted motion data is converted from quaternions to axis-angle pair type data. This is the only way of setting rotations to the Joint nodes in X3D.

B. FUTURE WORK

Using axis-angle pair data for rotation in X3D overloads the client program with additional computation. The first humanoid developed for the MARG project [Ref. 2] was capable of setting quaternion data directly to the joints without any additional conversion. This humanoid was developed in Java using the open-source *Java3d1.3.1* package [Ref. 17]. Tools other than X3D that supports quaternions might be required in future work when performance is an issue.

The humanoid Andy does not have a realistic geometry when compared with the previous laser-scanned humanoids. A realistic humanoid is required to increase the immersion in virtual environments. Therefore, the laser-scanned humanoids can be developed to simulate the motion captured by 15 MARG sensors. At this level, a cartoon type humanoid is sufficient for the MARG project.

The client-server application needs to be adjusted when 15-channel CIU is available. The MemoryUpdater class is responsible for reading motion data from the sensors. The current structure of the class receives motion data through three independent wireless serial adapters. A structure for one serial adapter needs to be developed by a modification to the current structure. Since all 15 sensors' data will be packed into one packet, a decoding method to the packet needs to be developed.

To obtain a better result from the Quest algorithm, a calibration process is required. The calibration results are obtained using a separate program. The results are displayed on the screen and not saved in a file. The results of the calibration process for each sensor are set to the Quest algorithm manually. This will be challenging to handle for a 15-sensor system. On the other hand, the QuestQuaternionProducer class, which implements the Quest algorithm, cannot be used for multiple sensors. That is, a new object of this class is created for each additional sensor. This class needs to be modified

to process multiple sensors. Adding data members to hold the calibration data for all sensors will solve this problem.

The current setting with the three WiSER2400.IP serial adapters meet the needs of this thesis. Each WiSER2400.IP is responsible for transmitting the motion data for one MARG sensor. In a 15-channel CIU setting, the size of the sensors' data will be more than the capacity of the current WiSER2400.IP (i.e. more then 200 bytes). Requesting an increase in the capacity of the WiSER2400.IP from the manufacturer will solve this problem. Otherwise, another wireless serial adapter is required. Another problem with the WiSER2400.IP is the battery consumption that limits the portability of the system.

The performance of the concurrent client-server application is not tested on the wide-area network (WAN). For example, testing can be conducted by setting the server program to run in the United States and the client programs to run in Turkey (more than 10,000 miles distance).

LIST OF REFERENCES

1. "ISO/IEC 19774 – Humanoid Animation (H-Anim200x)," [http://www.h-anim.org/Specifications/H-Anim200x/ISO_IEC_FCD_19774]. February 2004.

2. Bachmann, Eric Robert, *Inertial and Magnetic Tracking of Limb Segment Orientation for Inserting Humans into Synthetic Environments*, Ph.D. Dissertation, Naval Postgraduate School, Monterey, CA, December 2000.

3. Dutton, James Allen, *Developing Articulated Human Models from Laser Scan Data for Use as Avatar in Real-time Networked Virtual Environments*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 2001.

4. "Humanoid Animation," [http://www.web3d.org/TaskGroups/x3d/translation/examples/HumanoidAnimation/AllenDutton.wrl]. January 2004.

5. Sinav, Alper, *Analysis and Modeling of the Virtual Human Interface for the MARG Body Tracking System Using Quaternions*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 2002.

6. Vcom3D, Inc., "Human Characters," [http://www.vcom3d.com/Viewer.htm]. February 2004.

7. Schuster, M.D. and Oh, S.D., "Three-Axis Attitude Determination from Vector Observations." *Journal of Guidance and Control*, Vol. 4, No. 1, pp. 70-77, January-February, 1981.

8. Kavousanos-Kavousanakis, Andreas, *Designing and Implementation of a DSP-Based Control Interface Unit (CIU)*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 2004.

9. OTC Wireless, Inc., "Wiser Wireless Serial (RS232) Solution," [http://www.otcwireless.com/802/wiser.htm]. January 2004.

10. Cornelius, Berry, "Java versus C++," [http://www.dur.ac.uk/~dclObjc/Java.htm]. January 2004.

11. The Working Group for WLAN Standards, "IEEE 802.11 Wireless Local Area Networks," [http://grouper.ieee.org/groups/802/11/]. March 2004.

12. "802.11," [http://www.webopedia.com/TERM/8/802_11.html]. January 2004.

13. OTC Wireless, Inc., *802.11b Wireless Serial Port Adapter Wiser2400.IP User Guide*, January 2004.

14. "Extensible 3D (X3D) Humanoid Animation (H-Anim) Component," [http://mediamachines.com/X3D/spec_07_21_02/part01/components/hanim/index.html]. February 2004.

15. Web 3D Consortium, "Creating Open Standards for Communicating 3D," [http://www.web3d.org/vrml/browpi.htm]. January 2004.

16. McGhee, R. B., *MV4472 Advanced Physically Based Modeling Course*, Naval Postgraduate School, Monterey, CA, 2003.

17. Sun Microsystems, Inc., "The Source for Java Developers," [http://java.sun.com]. February 2004.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Dr. Rudy Darken
   Naval Postgraduate School
   Monterey, California

4. Prof. Xiaoping Yun
   Naval Postgraduate School
   Monterey, California

5. Don McGregor
   Naval Postgraduate School
   Monterey, California

6. Prof. Robert B. McGhee
   Naval Postgraduate School
   Monterey, California

7. Dr. Don Brutzman
   Naval Postgraduate School
   Monterey, California

8. Eric Robert Bachmann
   Miami University
   Oxford, Ohio

9. Andreas Kavousanos-Kavousanakis
   Hellenic Navy
   Athens, Greece

10. Deniz Kuvvetleri Komutanligi
    Personel Daire Baskanligi
    Bakanliklar, Ankara, Turkey

11. Deniz Harp Okulu Kutuphanesi
    Tuzla, Istanbul, Turkey

12. Arastirma Merkezi Komutanligi
Pendik, Istanbul, Turkey

13. Middle East Technical University
Department of Computer Engineering
Ankara, Turkey

14. Istanbul Technical University
Electric and Electronic Faculty, Department of Computer Engineering
Ayazaga, Istanbul, Turkey

15. Faruk Yildiz
Turkish Navy
Golcuk, Kocaeli, Turkey