ABSTRACT

Design and Implementation of Real-Time Software for Sourceless Full Body Tracking using Small Inertial/Magnetic Sensors

by Eric W. Montgomery

Practical full body tracking remains a difficult challenge related to the construction of immersive synthetic environment systems. Human body posture can be tracked in real-time using small, inexpensive inertial/magnetic sensor modules to track the orientation of individual limb segments and using the individual orientations to determine body posture. If the position of one point on the body is tracked, the position of all body segments can be determined. The software required for such a system is inherently complex due to its real-time distributed nature and the necessity to interface with varying sensor types and numbers. Furthermore, the software must be modular and flexible to ease enhancements and modifications.

This thesis describes the implementation and design of a software system capable of processing data from a minimum of fifteen inertial/magnetic sensors at an update rate in excess of 100 Hz. Sensor data can be collected by a wearable computer and submitted via wireless LAN to a fixed workstation. The workstation processes data and acts as a server for avatar animation data. Design of the system software was completed using object-oriented techniques and the unified modeling language (UML). Several conversion classes process raw data from the fifteen sensors before being processed by a quaternion based complementary filtering algorithm to produce orientation estimates in quaternion form. Estimates are sent to a graphical display unit through a local area network. At any point through the conversion process, data may be archived for data analysis purposes or reprocessing.
TABLE OF CONTENTS

1. INTRODUCTION......................................................................................................... 1
  1.1 INERTIAL/MAGNETIC POSTURE TRACKING ................................................. 4
  1.2 THESIS OBJECTIVE............................................................................................... 4
  1.3 CHAPTER SUMMARY ........................................................................................... 5

2. BACKGROUND ........................................................................................................... 7
  2.1 SOFTWARE ENGINEERING................................................................................. 7
  2.2 METHODS AND MODELS .................................................................................... 8
  2.3 SYSTEM REQUIREMENTS ................................................................................... 9
    2.3.1 REQUIREMENT GATHERING ..................................................................... 10
  2.4 SYSTEM DESIGN ................................................................................................. 10
    2.4.1 SYSTEM DESIGN DECISIONS .................................................................... 11
  2.5 SYSTEM VERIFICATION AND VALIDATION ................................................ 11
    2.5.1 VERIFICATION AND VALIDATION TECHNIQUES................................ 12
  2.6 IMPLEMENTATION, TESTING, AND DOCUMENTATION ........................... 13
    2.6.1 IMPLEMENTING THE SYSTEM.................................................................. 13
    2.6.2 TESTING THE SYSTEM ............................................................................... 13
    2.6.3 DOCUMENTATION ...................................................................................... 14
  2.7 SOURCELESS POSTURE TRACKING ............................................................... 14
  2.8 SUMMARY ............................................................................................................ 16

3. SYSTEM DESIGN AND IMPLEMENTATION..................................................... 17
  3.1 DESIGN DECISIONS............................................................................................ 17
    3.1.1 LANGUAGE SELECTION............................................................................. 17
    3.1.2 OTHER DECISIONS ...................................................................................... 18
  3.2 SYSTEM DESIGN ................................................................................................. 18
  3.3 IMPLEMENTATION OF THE SYSTEM ............................................................. 21
    3.3.1 IMPLEMENTATION OF THE WEARABLE COMPUTER................................. 21
      3.3.1.1 POLLING THE SENSOR MODULES .................................................... 21
      3.3.1.2 PACKAGING THE DATA ...................................................................... 21
    3.3.2 IMPLEMENTATION OF THE WORKSTATION......................................... 22
      3.3.2.1 ERROR LOGGER.................................................................................... 22
      3.3.2.2 DRIVER.................................................................................................... 23
      3.3.2.3 SETUP AND COMMANDWEARABLE ................................................ 23
      3.3.2.4 DATA CLASS.......................................................................................... 24
      3.3.2.5 WORKSTATIONCONTROLLER........................................................... 24
      3.3.2.6 WORKSTATIONDATAHANDLER....................................................... 25
      3.3.2.7 DATACONVERTER ............................................................................... 26
      3.3.2.8 CONVERSION CLASSES AND CALIBRATEDATA .......................... 26
      3.3.2.9 DATA SERVER ....................................................................................... 28
      3.3.2.10 INTERFACE........................................................................................... 29
    3.4 SUMMARY ............................................................................................................ 36

4. EXPERIMENTAL AND TESTING RESULTS ...................................................... 37
  4.1 DATA RETRIEVAL .............................................................................................. 38
  4.2 DATA CONVERSION........................................................................................... 38
  4.3 ARCHIVING DATA .............................................................................................. 39
  4.4 UPDATE RATE ..................................................................................................... 39
4.5 MULTICAST CONNECTION ................................................................. 40
4.6 INTERFACE .................................................................................... 41
4.7 SUMMARY .................................................................................... 41
5. CONCLUSIONS AND FUTURE WORK ............................................. 42
  5.1 SUMMARY .................................................................................... 42
  5.2 CONCLUSIONS ............................................................................ 42
  5.3 FUTURE WORK ............................................................................ 43
    5.3.1 IMPROVING THE CODE ......................................................... 43
    5.3.2 HARDWARE .......................................................................... 44
    5.3.3 BODY TRACKING AND VIRTUAL ENVIRONMENTS ............. 44
    5.3.4 GAIT MEASUREMENTS AND USER POSITION ............... 45
    5.3.5 INFLUENCE OF MAGNETIC FIELDS ON MARG SENSOR MODULES 45
    5.3.6 MECHANICAL CALIBRATION OF SENSOR MODULES .......... 46
    5.3.7 LIMB SEGMENT LENGTHS ................................................... 46
6. REFERENCES .................................................................................. 47
7. APPENDICES ............................................................................... 49
  APPENDIX A: PROCESS DESIGN OUTLINE .................................. 49
  APPENDIX B: RELATIONSHIP DIAGRAM .................................... 51
  APPENDIX C: REQUIREMENT SPECIFICATION ......................... 52
LIST OF FIGURES

Figure 2-1: Inertial/Magnetic Body Tracking System .......................................................... 15
Figure 3-1: Example Requirement Specification .................................................................. 19
Figure 3-2: Sample of Workstation Relationship Diagram using UML ......................... 19
Figure 3-3: Example UML Class Definition ...................................................................... 20
Figure 3-4: SetUp Dialog Box ........................................................................................ 24
Figure 3-5: Block Diagram of Quaternion-Based Orientation Estimation Filter ............. 28
Figure 3-6: Interface – Individual Sensor Settings Tab .................................................... 30
Figure 3-7: Interface – Data Source Tab ........................................................................... 31
Figure 3-8: Interface – Sensor Data Tab .......................................................................... 32
Figure 3-9: Interface – Digital Conversion Tab ............................................................... 33
Figure 3-10: Interface – Record Data Tab ...................................................................... 34
Figure 3-11: Interface – Sensor Settings ......................................................................... 35
Figure 3-12: Interface – Network Tab ............................................................................. 36
Figure 4-1: Achieved Update Rates ................................................................................ 40
Figure 4-2: Sample Avatar ............................................................................................. 41
Figure A: Data Flow Diagram ......................................................................................... 54
ACKNOWLEDGEMENTS

This thesis has taught me about things that I could not have learned in a classroom setting. As students, we often are caught up in figuring out how little we can do to get by with a certain grade. There is no room for error in a work of this magnitude. It takes 110% effort and 125% of your time. I will carry with me many memories and much knowledge from my six years at Miami University, but some of my fondest and most memorable moments will no doubt be from this work.

I owe thanks to many people for this accomplishment. First, I would like to thank my readers, Dr. Scott Campbell and Dr. Douglas Troy, who were patient and considerate in their comments and suggestions. I would also like to thank the professors and instructors of the Systems Analysis department for the knowledge they have imparted on me.

Gratitude is due to my friends and colleagues that have put up with many stress-filled days and still call me a friend, at least they say they do. Thank you (in no particular order) Chris, Ivan, Peter, Justin, Mike, the rest of the Friday night group, Shari, and Norm.

To my parents, who gave me life and have stood by me through everything I have ever gone through, I am much obliged. This is one of my biggest accomplishments, and I am grateful to have them to share it. And, of course, I could not possibly forget my younger brother, Christopher, who is more important to me as a friend and partner in crime than he realizes.

I offer special thanks and love to Christine, who is the light of my life. She has been there for me every second of every day since I started graduate school. More than anyone else, she knows how much this thesis and program has affected my life and changed it for the better. What she may not know is how much more of an affect she has had, and will always have, on my life. I cannot say enough how glad I am to have her.
Last, but certainly not least, I would not be writing this thesis without the constant guidance, patience, wisdom, and brilliance of Dr. Eric Bachmann over the past two years. He gave me the confidence to finish this work, and he pushed me when I needed it while still giving me the freedom to do the work as I saw fit (even if there was a better way). I will never forget the hours spent learning about quaternions, rotational matrices, Java, sensors, and so much more. I am a better person for having known and worked with him. I hope he knows how much I appreciate everything he has done for me – it did not go unnoticed. Thank you.
1. Introduction

The goal of body tracking is to track human posture and position in real time. Orientation estimates can be used to determine posture and used to drive the animation of an avatar. If the orientation and linear dimensions of all components of the user’s body are known, tracking the position of one point will allow animation and placement of the user’s avatar within a virtual environment.

The development of practical full body tracking is an integral part of synthetic environment (SE) research. It allows users to interact with a SE in a more natural manner than when using a device such as a mouse or joystick and thus enhances the sense of presence and illusion of realism within the environment. People interact with the world using their bodies. In order for users to be fully immersed in a virtual environment, their movements must be adequately tracked and displayed. The animation of an avatar using body tracking data allows users to view the actions of others in multi-user SE as well as see their own body within the SE. Making the posture and gestures of the user part of a SE also allows the SE to respond to their actions in an appropriate manner. Posture and gestures (specific motions or sequences of motion) are a highly important part of communication. Much of how people communicate with each other is done through non-verbal cues, and the ability to track and display this information can add to the believability of the environment. Currently, there are no practical real-time systems that can track and display the movements of multiple humans over a wide area. A system that can track and interpret human motion without being intrusive to the user is a goal of modern synthetic environments. [Sherman et al. 2003].

There are several requirements that can be used to determine if a body tracking system is adequate. They are sensor size, containment, completeness, accuracy, speed, resistance to interference, resistance to obstructions, range, wireless, and inexpensive. The size and weight of body-mounted sensors can be very important. If the device is large enough to be intrusive to the user, the user can feel encumbered and have difficulty maintaining a sense of presence. Containment refers to a system’s reliance on devices mounted on
walls. An ideal system is self-contained, relying only on user-mounted devices. The completeness of a system is its ability to track six degrees of freedom (pitch, roll, yaw, and position in x, y, and z). Six degree of freedom systems that can determine both orientation and location either directly or indirectly are necessary for most applications for user immersion into an SE. The accuracy of a system is a measure of its ability to maintain resolution and orientation to within certain limits. Speed measures how fast the system can react to and update the movements of the user. Resistance to interference is the system’s reliance upon line of sight. The system’s resistance to obstruction reflects how easily the system’s accuracy is degraded by outside forces. The range of the system is how far or how fast a user can roam from the “base” station. Wireless systems that can run on a small amount of energy for a long period (years) are ideal. Finally, the individual components of the system should be inexpensive. There are currently no systems meeting all these requirements. In fact, all available systems fail to meet at least seven of those characteristics [Foxlin 2002].

Most modern body tracking systems are “sourced.” Sourced systems must maintain a communication path between fixed transmitter or receiver stations and body mounted transmitters or receivers. The physical media of communication path can be one of several different types of waves in the electromagnetic spectrum. Currently available systems include sound, light, and magnetic fields. The downside of sourced systems is the need to maintain an unimpeded communication path between a fixed transmission or receiving station and the user. Noise sources can interfere with the path, causing faulty data or no data to be received. Additionally, in some systems the placement of any physical object between the user and the stations can result in shadowing and a loss of data or tracking information. This limitation does not provide a user with flexibility in terms of movement and can make it difficult to track multiple users in the same working environment. Another inherent disadvantage of sourced systems is range of use. The tracking area for sourced systems is typically limited to a living room sized area that must be extensively surveyed and calibrated for noise sources. These fixed areas are usually small because of the possibilities of interference and a drop off in signal strength [Bachmann 2000].
Sourceless body tracking systems do not suffer from the limitations of sourced systems that are described above. They do not require some type of communication with fixed stations in order to determine orientation and position. This eliminates both range limitations and interface problems.

Both sourced and sourceless body-tracking systems can benefit from a digital representation of the human body. To create a human avatar, consider the human body as a series of connected links with all links directly or indirectly connected to a single point, or root node. It is necessary to create a methodology capable of accurately describing the orientation of the links without sacrificing speed and efficiency. Given the orientation of each of the links, their length, and how they are connected, we can determine the orientation and position of any of the other links relative to the root. If the position and orientation of the root relative to an Earth-fixed reference frame can be measured, the orientation and position of all links can be determined relative to the same frame.

There are several methods commonly used to represent rigid body orientation. Three possible candidates include Euler angles, rotation matrices, and quaternions. Euler angles require only three numbers to represent orientation. However, Euler angles have singularity problems when the body segments assume particular orientations. Rotation matrix representation of orientation requires nine entries, to express a rotation. Three-dimensional coordinate vectors are transformed through matrix multiplication. This operation requires 15 scalar operations. All orientations can be represented using rotation matrices without singularities. Quaternions define an orientation using a four-tuple or four-dimensional vector containing three imaginary parts and one real part. Similar to vector angle pairs, the imaginary portion can be thought of as a three-dimension vector, and the real portion can be thought of as a scalar that is some way represents the angle of rotation. Three-dimensional coordinate vectors are transformed by augmenting them to create a four dimensional vector and subjecting them to quaternion multiplication. Transformation of a vector using quaternions requires 56 scalar operations; however, all orientations can be represented using quaternions without singularities. Additionally,
quaternions do not require the use of trigonometric functions and thus avoid the expensive approximation of these functions using Taylor series or some other method [Kuipers 1999].

1.1 Inertial/Magnetic Posture Tracking

Inertial and magnetic posture tracking is a sourceless body tracking technology. It uses magnetic, angular rate, gravitational (MARG) sensor modules attached to individual limb segments to determine the posture of the user. These sensor modules are lightweight and small. Each sensor module contains a triplet of three-axis sensors: magnetometers, angular rate sensors, and accelerometers. These sensors provide the data needed to determine body posture by allowing estimation of the orientation of each limb segment to which a sensor module is attached. The software for an inertial/magnetic system collects the data from the array of MARG sensor modules and runs it through several conversion and filtering processes to produce avatar animation data.

1.2 Thesis Objective

The main purpose of this thesis was to enhance the study of inertial/magnetic body tracking through the design and implementation of software to support a full-body tracking system. The main function of such an application will be to read raw data from the sensor modules and interpret their values through a conversion process. The requirements of the system are as follows:

- Save orientation data to a file at any point through the conversion process and reload the data for reprocessing,
- Archive sensor calibration data for the modules,
- Accept and process control commands,
- Incorporate a flexible, modular design to allow for future upgrades and changes,
- Reliably, efficiently, and quickly perform its operations so that the system is effective within a synthetic environment.

The particularly challenging aspect of this software is its real-time nature. If the system software is not properly designed, then the system will not produce timely accurate
orientation estimates and visual representations of the movements will be lagged and incorrect.

Appropriate software engineering techniques were utilized from the start of the project. The design was done using the formal systems and waterfall models. The mathematical and structured tendencies of the system itself made the models an obvious choice. At times, the real-time nature of the system and total design time did not allow all of the conventions of these techniques to be suitably met and new conventions were created to meet the end goals of the system. For instance, since data members of classes needed to be accessed as quickly as possible, the use of accessors and mutators was limited to avoid unnecessary function calls. Another example is the use of a hybrid engineering model. Typically, the waterfall method does not allow backward movement through design stages, but the hybrid with the formal systems method allowed all movement.

The overall design of the project was done using the Rational Rose 98 software package and the Unified Modeling Language (UML), and as such, adheres to object-oriented principles [Sommerville 2001]. In the beginning, rough sketches, diagrams, and loose terminology were used to form rigid project guidelines, goals, and requirements. The process design outline, found in Appendix A, had been used to create class and relationship diagrams. From these, code skeletons were generated and finally the code itself was written.

1.3 Chapter Summary

Chapter 2 of this thesis discusses background information regarding software engineering techniques that were used to implement the software for the inertial/magnetic body tracking system. It includes information on the hardware used for the system.

Chapter 3 is describes the design and implementation of software for an inertial/magnetic body tracking system. It contains a description of the system and its various classes, functions, and interface.
Chapter 4 describes experiments performed to validate the software. It demonstrates that the requirements of the system have been met.

Chapter 5 presents a conclusion of the thesis, a summary of the work accomplished, and future improvements and upgrades that could be made to the system and its hardware.
2. Background

This chapter describes the basic principles, processes, and ideals of software engineering. It discusses various methods for design, the qualities of a “good” software system, and the processes for implementation. This chapter also considers how to choose appropriate software engineering models, discusses the unified modeling language (UML), and includes a section on the hardware of the system.

2.1 Software Engineering

Software engineering is an important process for transforming an idea into a piece of working software. To accomplish this objective, it is necessary to consider a number of elements and requirements that affect the system. The best method for discovering the elements and requirements specific to an application varies based on the application itself. When dealing with a complex system, it is important to use commonly recognized models and methodology so future maintenance and upgrades can be done with ease. Moreover, choosing and utilizing the proper software engineering approach can enhance and simplify the software design process.

The process of software engineering defines the attributes of a so-called “good” software system. These generic attributes are

- Maintainability – the property of a piece of software that characterizes how easily the software can be modified to support future upgrades. Object-oriented design inherits this property by definition. Since the pieces of an object-oriented system are modular, changing or replacing one piece is relatively simple when compared to updating a piece of code that is deeply integrated and not constructed using separate components.

- Dependability – measured in failure rate or time between failures; depending on the application, it may be very important for a software system to run for months without failure. For a smaller piece of software, it may be important that it start and run to completion every time it is called. It can also be measured based on the cost of damage should the system fail.
• Efficiency – how well a system utilizes computing resources; wasted CPU cycles and poor memory management are the result of poor design. Ultimately, they result in failures and a decrease in dependability. Wasting cycles and memory are not conducive properties for such a system.

• Usability – how intuitive is the software for a user; software that is deficient in usability is ineffective because users will not want to operate the system [Sommerville 2001].

2.2 Methods and Models

A software engineering model is a high-level view of system development. It gives the engineer a blueprint of the components of a system and their relationships. Software engineering methods supply the engineer with a step-by-step process for developing software. Most methods follow the same basic process:

• Requirements analysis – define the parameters of the project through interviews and research, identify internal and external processes and procedures that affect the system

• System design – form a list of requirements and goals for the system; represent the system architecture using diagrams and an outline

• Verify the correctness of the requirements and diagrams with follow-up interviews, traceability studies, and other techniques

• Implement, test, and document the system for future maintenance and upgrades

The process design outline used for this project is attached as Appendix A.

There are many software engineering methods available that follow these steps. The two that best fit a real-time body tracking system are the waterfall and formal systems methods. The waterfall method implements a strict sequential approach to designing software. It uses the process discussed earlier, but the defining measure of the waterfall method is the inability to reverse direction through the process. Stages are performed as atomic entities. Once a stage has been completed, the engineer cannot go back to a
previous stage without restarting it completely. This is a good choice for systems with well-defined requirements [Sommerville 2001, Pressman 2001].

The formal systems method allows iteration between steps and movement in either direction through the stages. It uses formal mathematical representation of objects and data flow. This method is well-suited for use with highly scientific or mathematical requirements. The formal systems method assumes that both the user and programmer have scientific knowledge that they may apply to the design and implementation of the software. These methods were merged to create hybrid method that used some of the formality of the waterfall method with the technological notation of the formal systems method [Sommerville 2001, Pressman 2001].

2.3 System Requirements

The requirements of a system define what functions a system should perform and the constraints on its development. Requirements are broken into two categories: functional requirements and non-functional requirements. Functional requirements define specific operations and capabilities of the software. For example, the system might be required to archive data to file. Non-functional requirements are desired qualities of the software but cannot be measured by ordinary means. For example, a non-functional feature of the software might be an easy-to-use interface.

Interviews with users involved with the software reveals requirements related to what the system should do, how it should look, and goals and constraints for the system. In the beginning, the requirements may be rough sketches, diagrams, and loose terminology. From these requirements, rigid project guidelines, goals, and can be written in what is referred to as a “requirements specification document”. This document should eventually include almost all of the requirements of the final system. As this document is created, it should be reviewed many times by any software engineers working on the project and anyone who will be involved with the use of the software once it is completed. It should act as a guide for the remainder of the design process.
2.3.1 Requirement Gathering

Requirements are first gathered through interviews with users and other people who interact with the system. Each user requirement, typically provided in common language, is transformed into precise mathematical or scientific definitions in the design stage. The definitions are translated into code by the programmer in the implementation stage. The main advantage to using this method is that if the requirements are thoroughly investigated, then there should be few validation errors found when translating the definitions to actual code.

2.4 System Design

During the design stage of development, the requirements specification document is transformed from the user’s perspective to a programmer’s perspective. The specification document produced has a feel similar to pseudo-code. This document also includes more formal illustrations and data flow diagrams. Each sub-system (or class) is formally defined, including an outline of its functionality, its interface with other sub-systems, the data structures and algorithms it will use, and its constraints. Once this is completed, conventional methods are used to produce final object-oriented and data flow diagrams of the system.

The unified modeling language (UML) provides universal diagrams and symbols to design software that permits straightforward communication between two software engineers. The biggest advantage to using UML is future work done on the system will have a familiar set of diagrams and charts to work from, much like an electrical engineer’s wiring chart. UML is not a method; it is a way of representing objects and functions within a system while maintaining object-oriented practices. UML is often chosen over other languages of the same type because it is widespread and easier to learn and use [Fowler et al. 2000].
Rational Rose 98 is a software package that assists in designing diagrams and figures with standard UML. From diagrams, code skeletons for the project can be written. The diagrams make it easier to view and plan relationships between classes and entities in the system. Rational Rose is available from the Rational Software Corporation, and a demo version is available for download from www.rational.com. Microsoft produces a similar product, called Visual Modeler, which performs many of the same functions as Rational Rose. It does not provide the functionality needed to produce certain types of diagrams [Quatrani 1998].

The relationship diagram produced for this thesis can be found in Appendix B. It uses standard UML symbols. The lines with open diamonds at the end represent a composition relationship. The dashed lines with an arrowhead, ordinarily used to show dependencies, are used to show a data flow relationship. For example, consider the wearable computer. It transmits data received by the WorkstationDataHandler class, but it also receives data from the CommandWearable class. This diagram worked as a medium to alter the pseudo-code into actual code.

2.4.1 System Design Decisions
Design decisions are constantly being made throughout the entire process. These decisions affect what language will be used, how the components of the system fit together and interact, the layout of user interfaces, what algorithms to use, control structures, etc. There are several factors that can affect how design decisions are made, specifically, time to develop the system, real-time characteristics, and the hardware to be used.

2.5 System Verification and Validation
Verification and validation are two important steps in the software engineering process. By properly performing these tasks, a software engineer can save time and effort throughout the design, implementation, and testing stages. Verification involves procedures for ensuring the software is meeting the requirements in the specification
document. Validation is the set of procedures that ensure the software properly meets the needs of the client. The definitions of verification and validation sound similar, but they are not exactly the same. Verification focuses on the functional requirements while validation centers on the needs and desires of the client.

2.5.1 Verification and Validation Techniques

There are many techniques that resolve verification and validation issues. Some examples of possibilities are inspection, software testing, and algorithm analysis. Each of these individual elements serves as confirmation of a different aspect of the verification and validation process.

Inspection is the technique of evaluating the entire system, including requirement documentation, diagrams, source code, and documentation to determine how well they represent the system. Analyzing the portions of the system listed above may reveal mistakes or inconsistencies. This technique is employed throughout the system development process, as it may affect multiple areas.

Software testing is broken into two sub-categories: unit testing and system testing. Unit testing is performed on individual modules, classes, or any other sub-function of the entire system. This technique can uncover dependencies that would break the object oriented principle of encapsulation. System testing is executed by running specific sets of test data (usually meaningless) through the system and verifying the output against expected values. Testing reveals coding errors that are not caught during inspection or compilation.

Algorithm analysis involves using other techniques, such as asymptotic analysis, to determine whether the code can theoretically perform up to the required standards. This process may be used at any time during the development process, but it is best to use it before implementation. Should the analysis return poor results, it may be necessary to
rewrite portions of the code. If the code is part of a larger system, it may be more difficult to change after implementation than if it was analyzed beforehand.

2.6 Implementation, Testing, and Documentation

The next step is to refine the rough sketches, diagrams, and requirements of early stages into a structure containing pseudo-code and class diagrams. This structure is designed using a data-flow model and an object-oriented model. As its name implies, the data-flow model shows how data will flow through the system. The entity-relationship model shows the objects of the system and their interactions.

2.6.1 Implementing the System

Implementation is done using a design model. If the previous stages were done properly, programming the system is an obvious extension. The algorithms and diagrams provide a clear blueprint for how the code should be written and how it should perform.

2.6.2 Testing the System

The testing of the system is done during and after the code has been written. As described above, it is also considered to be a validation technique. The entire process of testing ensures that the code meets the requirements in the specification document. This phase is broken into four main pieces, which are all continuously performed throughout the testing process. These phases are:

- Unit testing – testing each component of the system as an atomic piece to make sure that individual algorithms and other small portions of code are behaving as expected.
- Module testing – testing groups of units (such as functions) and double-checking that units operate together correctly.
- Sub-system testing – this phase takes different groupings of modules that will work together in the final code and assures that they will perform properly.
• System testing – the final phase the programmer completes; it looks at the entire, completed system and uses sets of controlled test inputs and their associated outputs and compares them against known sets to see if they are the same.

• Acceptance testing – after the programmer has completed the first four phases, the client looks at the final product and determines whether or not the application is acceptable based on expectations and the requirements specification.

2.6.3 Documentation

The programmer is responsible for creating documents that will assist future support of the system as well as a user guide. Documentation should include brief descriptions, required parameters, and expected outputs of functions and algorithms. The diagrams created using UML in the design stage can be provided as a guide, as well. Comments throughout the code are also an important method of documentation. There are commonly accepted methods, for instance JavaDoc, that are recognizable by most programmers. Proper documentation is critical to the evolution and maintenance of a software system.

2.7 Sourceless Posture Tracking

An ideal body tracking system would allow the posture and position of an individual to be determined both indoors and outdoors. To capture the fastest motions, it should have an update rate of at least 100 Hz [Bachmann 2000]. If markers or sensors must be worn, they should not encumber a user and must be attached in a manner such that they remain in a fixed location relative to the limb segment. Tracking accuracy requirements for the orientation of limb segments varies according to the segment. In general an accuracy of 0.5 degrees is acceptable for most body segments although head tracking may require accuracies to within a tenth of a degree.
Figure 2-1: Inertial/Magnetic Body Tracking System

Figure 2-1 is a diagram of a possible sourceless body tracking system. The main components are fifteen inertial/magnetic sensor modules, and a data control unit (DCU) that gathers all sensor data and transmits it via a wireless LAN to a fixed workstation for further processing. In the figure, user body posture is determined by attaching an inertial/magnetic sensor to individual limb segments. The sensors provide data that may be used to produce an estimation of the orientation of each sensor module relative to an Earth-fixed reference frame. The MARG (Magnetic, Angular Rate, and Gravity) modules contain nine sensing axes consisting of three orthogonally mounted angular rate sensors, three orthogonally mounted accelerometers and three orthogonally mounted magnetometers. Given the rotational offset between the reference frame of the limb segment to which a module is attached and the Earth-fixed reference frame, the orientation of the limb segment can be calculated. The posture of an avatar can be set through the addition of rotated limb translation vectors.

In the system depicted in Figure 2-1, there are three main pieces of hardware. The first piece of hardware is an array of up to sixteen MARG sensor modules. The individual sensors inside the modules include three small accelerometers, three magnetometers, and three angular rate sensors mounted orthogonally in triads. Each sensor triad measures a vector relative to the coordinate frame of the sensor module. The accelerometers measure the angle between the sensor’s frame and the gravity vector, the magnetometers measure the angle between the sensor’s frame and the local magnetic field, and the angular rate
sensors measure the speed of angular rotation in three dimensions. The sensors, working in concert with a complementary filter, yield an estimate of the orientation of the sensor module relative to an Earth-fixed reference frame. Using this information, the orientation of a limb segment to which a sensor module is attached can be determined. Given the orientation of all segments, body posture can be ascertained [Bachmann et al. 2001].

The second piece of hardware is a wearable computer or data control unit (DCU). This unit must be able to receive the data from up to sixteen sensor modules at a rate of at least 100 Hz. It also must have the processing speed to send the data over a network connection as it is read.

The third system hardware component is a base workstation computer. The workstation must have enough processing power to perform several functions. It must receive data over a wireless datagram connection from the wearable computer, and it must have enough speed to estimate orientation using the raw sensor data and send the estimates over a multicast connection to another piece of software.

2.8 Summary
This chapter discussed the principles, processes, and ideals of software engineering as well as different hardware used for an inertial/magnetic system. It showed how various methods for design and the processes for implementation could be used to create a “good” software system. This chapter also considered how to choose appropriate software engineering models and use the unified modeling language (UML) with Rational Rose software. Use of the aforementioned models and methods to build the system for this thesis is explained in the next chapter.
3. System Design and Implementation

This chapter describes how software engineering processes were used to create software for an inertial/magnetic body tracking system. It applies the principles discussed in the previous chapter to design and implement the software system. This chapter also includes an explanation of the code on the wearable computer and the workstation, how the objects and components fit together, and how data flows through the system. Various diagrams are included to assist the reader.

3.1 Design Decisions

Throughout the design and implementation process, decisions were made that affected the final product. The primary decision was the choice of programming language. Other design decisions included the use of error logging and exception handling.

3.1.1 Language Selection

Many factors affect the choice of programming language. This system in particular had aspects that influenced the available language options. These aspects were its real-time nature, the required user interface, the ease with which future maintenance on the system could be performed, and the amount of time available to develop the software.

Java was chosen as the language for the primary code on the workstation. Sun provides a full, easy to use, API that contains information about functions, data members, expected inputs and outputs, parent classes, child classes, and other information, available from its website. JBuilder7, an application development environment, provides functionality for designing and modifying a graphical interface. Finally, there was a general feeling that given the time constraints and the object-oriented character of the system that Java would provide the necessary functionality.

There was concern, however, that Java would not be fast enough for a real-time application. The timing of Java’s garbage collector cannot be controlled, but it can be delayed. To avoid wasting memory, as many objects as possible are instantiated in the
constructors of various classes instead of in the functions they are used. By doing this, the garbage collector runs less frequently.

3.1.2 Other Decisions

Instead of writing error messages to a console window, the application was designed to write the errors to a file. The file is opened at the onset of the program, and saves and closes when the program exits. After the program is run, the user may check the error logs to see what, if any, errors occurred. This was seen as a superior way to track errors since the error file still exists after the console window is closed.

Java is designed for simple exception handling, and this feature was exploited throughout the system. Proper exception handling increases the availability (how long the system can operate continuously) and reliability (how many fatal errors occur) of the system by dealing with errors in a way that allows the system to continue running. Often, the exceptions that are caught cause a dialog box to open that inform the user of a mistake. Exceptions that will not cause the system to crash are written to a file for future maintenance. In some places, default values are used in place of incorrect values.

3.2 System Design

The real-time nature of this system calls for accurate and concise design since any flaws may cause magnified delays during run time. The structure of the waterfall method complemented by the terminology and design of the formal systems method lent itself to the design of a software system for body tracking. This system was designed using a hybrid of the waterfall and formal systems methods.

The first stage of the design process calls for interviews with the users and others who interact with the system. The interviews revealed requirements of system functionality, its appearance, and goals and constraints of the system. From rough sketches, diagrams, and loose terminology, rigid project guidelines, goals, and requirements were written in a requirements specification document. These requirements and goals included hardware
requirements, software and hardware interaction, and user interface design. This
document is included as Appendix C. An example of a requirement specification used in
designing this particular system is found as Table 3-1.

<table>
<thead>
<tr>
<th>Function Name:</th>
<th>ConvertRawData</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>Converts data from raw integers to voltages</td>
</tr>
<tr>
<td>Inputs:</td>
<td>9 integers ranging from 0 to 32768</td>
</tr>
<tr>
<td>Outputs:</td>
<td>9 floats ranging from 0 to 5</td>
</tr>
<tr>
<td>Location:</td>
<td>Workstation</td>
</tr>
</tbody>
</table>

Figure 3-1: Example Requirement Specification

Designing the system follows the completion of the requirements specification. Each of
the functions of the specification is broken down into classes or objects. During this
stage, diagrams are used to represent relationships between these objects. Rational
Rose98 provided an interface for using UML notations and creating diagrams. Figure 3-2
demonstrates the how they are used to produce a relationship diagram. The entire
diagram, found in Appendix B, was used to implement the system for this thesis.

Figure 3-2: Sample of Workstation Relationship Diagram using UML
The rectangular boxes of the figure represent classes or objects. Lines with diamonds are used to show aggregate (composition) relationships between classes. Inheritance would be represented by an arrow with a triangle. This system does not use inheritance. The numbers near the endpoints of the relationship lines tell the engineer how many objects belong to any particular class. For example, there are six quaternions in each DataClass object. The dashed lines, which typically represent dependency, are used in this diagram to represent data flow. A note appears near the top of the diagram to alert future engineers about this and other differences.

Figure 3-3 is an example of a class definition using UML. The rectangle is divided into three smaller boxes. The top box holds the name of the class, in this instance, Quaternion. The middle box shows the data members of the data. If any members are private, they are denoted on the left side by a small lock. The bottom box displays the functions of the class. Similarly, any private members methods can also be denoted with on the left side with a small lock on the left side.
3.3 Implementation of the System

The system is broken into two main segments. The array of MARG sensor modules, the wearable backpack computer, and the software connections between them constitute one segment of the system. The other segment is the main software on the workstation computer. This portion receives the data sent from the wearable computer via a socket connection, converts it, and transmits it to a multicast group.

3.3.1 Implementation of the Wearable Computer

The wearable computer executes two basic functions. It polls the sensor modules for data, and it packages and sends the data to the workstation through a datagram connection. A datagram connection is used in place of a stream connection. Stream connections are considered reliable due to the use of acknowledgement packets, but the acknowledgement packets also require more time to send. If a new packet is sent while a bad packet is being resent, then data will be received by the system in the wrong order. The code for the wearable computer was written in C++. The interface between the sensor modules and the wearable computer built by National Instruments required C++.

3.3.1.1 Polling the Sensor Modules

The wearable computer reads in polled data from National Instruments data cards that are installed on it. These cards do the actual polling of the sensors and act as an intermediate between the modules and the wearable computer. The card performs polling activities 20,000 times/second and buffers the data until requested. The wearable computer then polls the National Instruments cards at a rate specified by the user (typically 100 times/second). The data received from the cards is an average of the data received since the previous sample was taken.

3.3.1.2 Packaging the Data

Using the data from the cards, the wearable computer creates a packet. The packet sent to the workstation is a series of integers. The first integer is the time in milliseconds since
the last reading from the sensors (delta-T). The next nine integers sent are the reading from sensor module 0’s angular rate sensor (X-Y-Z axes), its accelerometer (X-Y-Z axes), and its magnetometer (X-Y-Z axes). The next nine integers are the same data from sensor module 1, then module 2, and so on, until all of the sensor modules have been packaged. These sets of nine integers represent the voltage output of the individual sensor modules. A maximum integer is used on the workstation side to convert the integers to primitive float values, which correspond to the voltage outputs of the sensor module.

3.3.2 Implementation of the Workstation

The workstation has the largest section of the code written for the body tracking system. This code receives data from the wearable computer, transforms it into offset quaternions, saves the data, calibrates sensors, receives and interprets user commands, and forwards the data to a multicast connection where data display units can pick it up and animate an avatar. It has several classes that interact with each other and network connections. The following classes describe the classes and some of the functions of the workstation.

3.3.2.1 Error Logger

The class ErrorLogger is a member of every class on the workstation. The class receives error messages and writes them to a file. In this manner, the errors are logged so that the user may debug without the use of the console window. Its constructor takes a String as a lone parameter that is used as the file name of the error log. It also has two functions logError(String), logError(Exception). The logError(String) function takes the String and writes it to a file as a byte stream. There is a menu option, “Convert File -> Convert,” that will display the error messages in the console window at the user’s discretion. The logError(Exception) prints the stack trace as a String to the file. The file can be opened up and read as a typical text document. In this manner, errors can be saved for future use. Both methods write to the same file, but logError(Exception) is more commonly used because it allows the user to open up and read the file without the use of the interface.
3.3.2.2 Driver

The main method of the workstation code is found in the Driver class. Driver first calls SetUp to initialize the code.

3.3.2.3 SetUp and CommandWearable

SetUp requests the IP address and port number of the wearable computer and the number of sensor modules to be used from the user using a dialog box (Figure 3-4). It creates an instance of CommandWearable (CW), gives it the IP address and port number for the wearable computer, and calls the appropriate method, openConnectionToWearable(). This opens a connection to the wearable computer that will be used throughout the rest of run-time. SetUp next calls the findNumberOfSensors() method of CW. CW makes a request to the wearable computer for a single packet of data. Since this is over the stream connection, the packet is guaranteed to arrive in order and intact.

A single packet of data from the wearable computer arrives as a series of integers. Once the delta-T value is removed from the stream, the remaining integers should be divisible by nine since each sensor outputs nine integers. If the remaining number is not divisible by the nine, the code outputs an error message and logs the error through ErrorLogger. By dividing by nine, the number of sensors can be verified.

Once SetUp has the number of sensors, it retrieves the file names for the sensor calibration data. By default, these values are saved with specific names in a folder called “Files” that is located in the same directory as the class files for the code. If the folder does not exist, the code cannot find it, the files have been misnamed, or some other error occurs, it initializes the calibration data to zeroes and saves them with the default names in C:\Temp. The last function performed by SetUp is to create the array of DataClass objects.
3.3.2.4 Data Class

DataClass is a container class. One instance is created for each sensor module. It does not have any functions, and its data members are all public. Since Java passes a class into a function by reference, never by copying it, the instances of DataClass are shared by the rest of the classes in the code. This accomplishes two things. First, it saves space by having only one instance of the class in memory. Second, it saves on the amount of data that needs to be passed from class to class during archives and conversions. By saving the amount of data passed, time is also saved [Larman 2000]. Once the instances of DataClass are created, they are copied into CW and used throughout the rest of the code.

3.3.2.5 WorkstationController

After SetUp has finished, Driver creates an instance of WorkstationController. WorkstationController controls execution until the program is terminated. As parameters to its constructor, it takes a reference to the DataClass array, a String array of the sensor calibration file names, and a reference to CommandWearable, created by SetUp.
WorkstationController has two functions, its constructor and runProgram(). The constructor creates instances of a DataConverter, a DataServer, a WorkstationDataHandler, and an Interface. It then calls runProgram() which is the infinite loop in which the program will run. The infinite loop checks to see if the user has chosen a source from which to receive data (either raw data from the wearable computer or archived data from a file). This choice is made through the interface (more about class Interface can be found below). If not, it continues to loop waiting for the user to choose from where input will come.

3.3.2.6 WorkstationDataHandler

The methods of WorkstationDataHandler are responsible for handling data from the wearable computer and transferring it to the DataClass. It also calls the conversion methods of DataConverter (detailed below). The class has a constructor and four methods. The constructor receives a reference to DataConverter and DataServer objects created by WorkstationController. The constructor calls createDatagramSocket(), which sets up the datagram connection to the wearable computer using the IP address input by the user and a predetermined, arbitrary port number of 7984. If the socket cannot be connected on the first attempt, the code sleeps for 5 seconds and retries. If it fails the second time, the connection will not be created, and the boolean flag datagramConnectionMade will be set to false. This flag is used in the handleData() method. This method makes the appropriate method call to DataServer based on the dataSource flag, which can be changed via radio button by the user on the “Data Source” tab of the interface. If the source of the data is raw data from the wearable computer, then WorkstationDataHandler makes a call to its own method, receiveData().

The receiveData() method reads data from the datagram socket connection created by the createDatagramConnection() method. If the connection has not been made (that is, datagramConnectionMade is false), then it makes a call to createDatagramConnection() to create a connection. As the data is read from the socket, it is placed in rawData via a reference to the DataClass array through DataConverter.
The last method of WorkstationDataHandler is a toggling function, toggleRecordData(int recordFlag). It receives a flag from class Interface when the user changes the source of the data. It then changes its own flag and makes a call to DataServer to provide it with the new information.

3.3.2.7 DataConverter

The DataConverter is an aggregate class, composed of the various conversion classes. It is instantiated by WorkstationController, passed by reference to the WorkstationDataHandler, and controls the conversion of the data. Its constructor creates instances of a VoltageConverter, an RPSGGConverter, a Filter, and an Offsets. Its only method makes calls to the convert() method of VoltageConverter, the convert() method of the RPSGGConverter, the filter() method, and the offsetQuaterion() method of its constituent objects. However, it only makes calls to the components as necessary. Thus, if the user has chosen to read in data from a file that has already been run through both convert() functions, a flag will tell DataConvert to begin at Filter.filter().

3.3.2.8 Conversion Classes and CalibrateData

Each of the conversion classes (VoltageConverter, RPSGGConverter, Filter, and Offsets) serves a single purpose so that if new conversion models are to be tested in the future, it will be easy to replace the current ones. Each receives only a reference to the DataClass array in its constructor. This reference is used by each to update the appropriate array within the appropriate object as it makes its calculations. VoltageConverter reads from the rawData array and writes to the voltages array of the same DataClass object. It converts the raw integers from the wearable computer into the appropriate voltage output. RPSGGConverter translates the data from voltages and places it in the primitive float array, RPSGG (Radians Per Second, G’s, Gauss – physics measurements of velocity, acceleration due to gravitational forces, and magnetic field strength, respectively).
RPSGGConverter uses an instance of CalibrateData to record the null and scalar values used in transform the raw data to useful units of measurement. CalibrateData is a series of public float arrays that have one index corresponding to each sensor. The values for the arrays are saved in files and are loaded at the onset of the program. The values may be changed at any time by the user at the interface, by either entering new values or choosing a new file.

The filter is one of the most important pieces of the entire system. The class Filter uses the physics measurements generated by RPSGGConverter.convert() to create a raw quaternion that represents an estimate of the orientation of sensor module relative to an Earth fixed reference frame.

The MARG sensor filtering algorithm produces accurate orientation estimates by taking advantage of the complementary natures of the three types of sensors in the units.

The manner in which the data from the different types of sensors is combined can be described as follows: The accelerometer triad output is normally averaged for a period of time in order to measure the components of the gravity vector or the local vertical relative to the reference frame of the triad. Determination of the relationship of the coordinate frame of the sensor unit to this known vector allows estimation of orientation relative to a horizontal plane. Similarly, the orthogonally mounted triad of magnetometers measures the local magnetic field vector in body coordinates allowing determination of rotation about the vertical axis. Thus, combining magnetometer data with low frequency accelerometer data provides a method for estimating the orientation of a static or slow moving rigid-body. Alternatively, assuming the initial orientation is known, integration of the output of a triad of orthogonally mounted angular rate sensors provides another method of estimating orientation. If the rate sensors are susceptible to noise or bias effects, as is the case for the small low cost sensors discussed here, errors will render these estimates useless after a short period. In dynamic applications, MARG sensor filtering algorithms combine high frequency rate sensor orientation information with low-frequency accelerometer and magnetometer data in a complementary manner to produce
accurate orientation estimates in real-time. Figure 3-5 is a block diagram of the complementary filter implemented as part of this research. A more indepth discussion of the theory and derivation of the filter can be found in [Bachmann 2003].

The class Offsets uses the raw quaternion generated by the digital filter and a reference quaternion to yield an offset quaternion, which is later sent to a display unit.

### 3.3.2.9 Data Server

The class DataServer handles outputting data to a file or broadcasting it via a multicast connection. DataServer takes a reference to the DataClass array as its lone parameter. Its constructor initializes the connection to the multicast address as well as files’ input and output streams. After the data has been converted by DataConverter.convertData(), the recordData() and transmitOrientationData() methods are invoked by WorkstationDataHandler. The user chooses the name of the file to which to write specific data, which is set (along with a flag) in the DataServer class. When the method is used, DataServer runs through the flags and writes data to files, if requested. The transmitOrientationData() function sends the offset quaternions with body limb segment
flags over a multicast connection to any computers listening. Applications running on these computers have the functionality to animate an avatar.

DataServer also has functions to load data from a file. Five methods read files containing different types of data and place the data in the appropriate place in the DataClass array. They are called when a user chooses a data source different other than new data from the wearable computer.

3.3.2.10 Interface
At over 4000 lines, the Interface class is the largest class of the entire software system. It was built using JBuilder7’s visual interface design capability. The class constructor takes a WorkstationDataHandler, a CommandWearable, and the number of sensors as an integer primitive. Through the WorkstationDataHandler and CommandWearable objects, the Interface can access each of their component objects. The Interface has seven panels. They are broken down by functionality. The panels are named Individual Sensor Settings, Data Source, Sensor Data, Digital Conversion, Record Data, Sensor Settings, and Network. Figures are provided for each of the tabs, and several menu options are described after the descriptions of the tabs.

The Individual Sensor Settings tab (Figure 3-6) allows the user to change scalars, nulls, gravitational and magnetic vectors, the filter time constant (Tau), the filter gain, and the magnetometer weighting. It allows the enabling or disabling of magnetometer weighting as well as a low pass digital filter for angular rate sensor data. The various pieces are tied to other classes in the system. The scalars and nulls change the values in CalibrateData. The remainder are stored in the data class and used in the Filter. Any values the user inputs and attempts to apply are error prone. These errors are caught in try-catch blocks that throw a number format exception. If the string in any of the text boxes cannot be converted to a number, an exception is thrown, and none of the changes is applied.
The next tab, Data Source (Figure 3-7), allows the user to choose where to read source data from. The data may come from either the wearable computer or a file. A file must be selected before the system will allow a user to choose a file to read. Once applied, Interface makes the appropriate changes in the Data Server and WorkstationDataHandler classes.
From the Sensor Data tab (Figure 3-8), the user can link the sensor number and limb segment. The Interface sets a flag in the DataClass, which is sent as a flag to the display unit over the multicast connection. The user may change the length of the limb segment using this tab, but the functionality for storing, changing, and utilizing the limb segment lengths has been implemented. When the “Output File Select…” button is selected, the user may choose a file to which to write the sensor’s data, and whether to record the data with the “Recording?” button. Finally, the user may select a new calibration file for a sensor but must apply the new values on the Individual Sensor Settings tab.
The Digital Conversion tab (Figure 3-9) is the simplest of the seven. It allows the user to change the boundaries on the range of the raw integers coming in from the wearable computer and the range of the voltages those numbers represent. Changes made here are applied to the VoltageConverter and RPSGGConverter.
Figure 3-9: Interface – Digital Conversion Tab

On the Record Data tab (Figure 3-10), the user may set up the types of data to record to a file. No default files are enabled, so the user must select a file name for a given file type from the drop down box labeled “Select File” before attempting to enable recording. If a file is not set up, then a message dialog box appears with a brief error message instructing the user to choose a file before continuing.
The Sensor Settings tab (Figure 3-11) operates in a manner similar to the Individual Sensor Settings tab. The only true difference between the tabs is that changes applied on this tab are applied to all the sensors, while the changes applied on the Individual Sensor Settings tab affect only the chosen sensor. This tab was included primarily for future use when the scalars, null values, and other calibration values will be the same for all sensor modules.
The last tab is titled Network. This tab is primarily used to provide the user with information about the wearable computer, multicast addressing, and the workstation computer. The user may change the multicast address and/or port number on the fly. The Apply New Values button calls a function in the DataServer to close the previous connection and instantiate a new one. Figure 3-12 shows the Network tab.
3.4 Summary

This chapter discussed how software engineering processes are used to create an inertial/magnetic body tracking system. The theoretical principles from Chapter 2 were expanded and applied to design the application. The design stage is the longest part of engineering a significant software system, and if done correctly, implementation follows naturally.
4. Experimental and Testing Results

This chapter validates that the system designed and implemented for this thesis meets all the requirements agreed upon by the engineer and client. Provided are various tests and analysis that were performed using a standard desktop computer. The computer had a 1.7 GHz potential, 256 MB RAM, and ran under the Windows XP operating system. Since a full array of sensors was not available at the time of testing, the code for the wearable computer used a static set of integers to send to the workstation to mimic additional sensors.

Each section below is an examination of a requirement set forth by the requirements specification document (Appendix C) that must be met for the system to be accepted. These requirements are:

- The raw data, received via a network connection at the workstation, will run through several processes to convert it into usable quaternion form. The processes written for the thesis will receive integer data, convert it to raw voltages, scale the voltages into usable units of measure, use the data to estimate sensors orientation, and then finally apply offsets to the quaternions to produce limb segment orientations.
- At any point during the process, functionality will be provided to save the data to a file. Archiving the data may or may not stop the data from being sent to the next function, at the user’s option. Similarly, all conversion processes must be equipped to receive data from a previous function or archived data.
- Sensor calibration will be done by another process that will save the data to a file. The software for the thesis will be capable of reading the file, loading the data, and applying it where applicable.
- An update rate of 100 Hz is expected, and the software will process data to produce orientation estimates for up to sixteen sensors simultaneously. Testing will be performed with a minimum of two sensors.
- Orientation estimates in quaternion form will be made available via a multicast network connection for avatar animation.
• Provide a reasonable interface that allows the user to send commands to the software, load new calibration data, and load and/or save archived data.

4.1 Data Retrieval

The class WorkstationDataHandler is equipped to use Java’s DatagramSocket and DatagramPacket classes to receive packets of data from the wearable computer. These packets are read and interpreted throughout the system (details of this process can be found in Chapter 3). The functionality of receiving packets of data through a UDP connection was tested by sending various packets of data to the workstation from a testing node. The following test packets of data were sent:

• A typical packet, with proper formats according to the protocol
• A packet with negative integers
• An empty packet
• Packets with too few or too many integers
• Packets not following protocol

The system is able to handle and interpret packets formatted according to the protocol without difficulty. Empty packets or packets with too little data cause the filter to throw a singular matrix exception, which is caught by the WorkstationDataHandler and logged for future reference. If the incoming data packet is too large, a similar exception is caught and logged along with a message to check the wearable computer for errors. Any other packet not conforming to protocol will also be ignored and logged.

4.2 Data Conversion

The data conversion process begins with raw integers and finishes with an offset quaternion through the conversion classes. To test the correctness of this process, a packet of data was created using a set of values that produced a known result. The test packet was a set of specific values that produces the offset quaternion \[0, 0, 0, 1\]. The
packet was received by a network connection, run through the conversion process, and produced the expected result over a multicast connection.

4.3 Archiving Data

Data archiving is done entirely through the DataServer class. The user chooses the file name through the Interface, selects how many recordings should be made, and when to begin. The system opens the file and writes data at the desired rate. The functionality is also available to replay the file (making the proper conversions) and convert the file from binary to ASCII.

Testing the archiving process was done in the following manner. First, the system was run in normal mode (receive data from the wearable computer, convert it, and send it via a multicast connection). As movements were made, the data was saved for each option at the same time. Thus, if the archiving and reading is done properly, each of the saved files should show similar movements (movements may be slightly different between files based on sensor calibration and offset calculations). If an invalid file is loaded, an error is logged, and no conversion takes place. A valid file may then be loaded by the user.

4.4 Update Rate

The update rate of the machine is expected to exceed 100 Hz with a minimum of fifteen sensors in use. Since at the time of this writing the requisite number of sensor modules was not available for testing, simulated data was used. The code on the wearable computer was set up to poll the available sensors for real data, and the remainder of the data was the same fixed set of integers used to test the validity of the conversion process. Figure 4-1 shows the update rates achieved during the testing process. Trials were run by sending data for two minutes and recording the number of updates received for each second. Faster update rates are possible by modifying the code that initializes the sensor modules on the wearable computer.
<table>
<thead>
<tr>
<th>Number of Sensors</th>
<th>Number of Trials Run</th>
<th>Mean</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>120</td>
<td>101.95</td>
<td>0.7316</td>
</tr>
<tr>
<td>6</td>
<td>120</td>
<td>100.77</td>
<td>0.7640</td>
</tr>
<tr>
<td>11</td>
<td>120</td>
<td>100.73</td>
<td>0.6447</td>
</tr>
<tr>
<td>16</td>
<td>120</td>
<td>100.79</td>
<td>0.7206</td>
</tr>
</tbody>
</table>

Figure 4-1: Achieved Update Rates

Testing was also performed on the workstation computer alone. To test the speed of the conversion process without the wearable computer, a testing node was created to send a series of integers representing the data from 16 sensors. This data was received by the workstation, converted, filtered, and orientation estimates were produced and multicast. An update rate of over 300 Hz was achieved for this standalone process.

4.5 Multicast Connection

The multicast connection sends data packets of quaternions and integer arrays for multiple display units to read. The quaternions represent rotation, and the array is an array of integer flags, each of the numbers 0 through 15 representing a different limb segment. A simple multicast receiver was created to verify that the connection was behaving as expected. The receiver was designed using Java3D, and it displayed a simple avatar shaped in the form of a cube. This avatar is shown in Figure 4-2. The data being sent from the workstation was printed to the console window, and the data received at the multicast node was printed to a separate window. The values sent and the values received matched, therefore, the multicast connection works properly. If an error occurs during any point at the workstation, no data is sent. An avatar was not part of the scope of this thesis, so no error checking was performed on the testing node side of the connection.
4.6 Interface

The Interface was the most difficult piece of the system to test because of its massive size. From this class, all other classes can be manipulated. Testing was performed to ensure that sensor setting updates were accurately updated, calibration files were properly loaded and saved, retrieval of data from multiple sources was possible, and archiving data at any point through the conversion process occurred. Most testing was performed through print statements to the console window and dialog boxes. As updates were executed, the print statements traced the path of the data flow and ensured that the changes were successful. Files were created that had known values, and different files were loaded for different purposes to see that the information was replaced with new values. For archiving, the data was saved to a file and reloaded (see above). It was also converted into an ASCII file and manually inspected for errors. Finally, the data was archived to multiple sources at the same time and verified as above.

4.7 Summary

Validation and verification of a system is an important process through the design and implementation of a system. It helps ensure that the system is built according to specification and that it meets the needs of the client. This chapter discussed many tests that were run to perform this procedure. Errors were found and corrected through this process, and the system emerged as an able, working application.
5. Conclusions and Future Work

5.1 Summary
The process of estimating body orientation and position is an important endeavor for synthetic environment research. A well-designed system can enhance users’ experiences with such an environment by allowing them to interact more naturally and suspend disbelief. An animated avatar allows users to view themselves and each other, and it provides an avenue to body language, a vital mode of communication. Such a system must be non-intrusive to the user, process quickly, accurately, and efficiently, and react in real-time. This thesis described the design, implementation, and testing of a system intended to reach these goals of modern body tracking.

The implementation is capable of processing data from a minimum of fifteen inertial/magnetic sensors at an update rate in excess of 100 Hz. Sensor data is collected by a wearable computer and submitted via wireless LAN to a fixed workstation. The workstation processes data and acts as a server for avatar animation data. Design of the system software was completed using object-oriented techniques and the unified modeling language (UML). Estimates are sent to a graphical display unit through a local area network.

5.2 Conclusions
This research represents the second major system developed using the MARG sensor technology. It is the first to contain networking nodes to transmit data over wireless connections, and it is the first written in multiple programming languages. The system performs at the desired rate of 100 Hz with sixteen sensors and the functionality to convert and archive data. The data archiving will be used for post-processing of the sensor and animation data as well as the reanimation of an avatar.

Since this system is sourceless, it is only limited by the range of a wireless LAN connection. The sensor module is lightweight, and therefore, barely noticeable by the
user. The methodology of three-axis accelerometers, three-axis magnetometers, and three-axis angular rate sensors used in combination with the complementary digital attitude filter provides a fast and accurate system.

5.3 Future Work

The system described by this thesis is part of ongoing research. There are expansions and modifications that can be made to improve the overall functionality and usability of the system.

5.3.1 Improving the Code

To further improve the speed of the workstation, other options should be considered. One such alternative is to compile the Java source code into a native machine code instead of byte code. Compiling Java into machine code requires a special software package, for instance, VisualAge or Visual Café. This process may decrease the amount of overhead of certain Java features like the garbage collector.

The class Interface is large, at over two-thirds of the entire workstation code. This class should be broken up into subclasses, where each subclass would be a panel. Thus, modifications to the design or code of a single panel could occur smoother than the current system. Also, work on creating arrays of text boxes and labels in conjunction with JBuilder7’s interface designer might produce an interface that is easier to maintain.

The file writers could be written as a buffered stream, instead of the basic input and output streams.

The file writers could be written as a buffered stream, instead of the basic input and output streams. The buffered input and output streams store bytes read or written to a buffer. When it is optimal for these bytes to be stored or retrieved, a standard input stream or output stream is called to read or write the read from or to a file, respectively. Reading and writing to a file is an expensive operation in terms of time. By buffering and
doing large reads and writes instead of incremental, the overhead time to perform these processes is reduced [Sun 2001].

5.3.2 Hardware
This system can also be improved by upgrading the hardware and modifying the sensor modules. The wearable and workstation computers can be upgraded with faster processors and more RAM, thus decreasing the running times of the program. Since the size of the code is a considerable factor, upgrade requests are reasonable.

At the time of this writing, work is near completion to decrease the size of the sensor module, as well. The new modules are approximately a 1” cube and are less intrusive to the user. This advancement is a step toward an optimal wristwatch sized module. New components for the modules are continuously being developed. As advancements are made in the individual magnetometers, angular rate sensors and accelerometers, the overall sensor modules becomes more accurate and faster.

5.3.3 Body Tracking and Virtual Environments
The foundation has been laid for a full body tracking system with sixteen MARG sensors. The software system designed here paves the way for future work in tracking multiple users over a wide area; yet more work needs to be done to create a synthetic environment capable of handling many users. The users will participate by wearing a set of MARG sensors and possibly a head mounted display to view the environment.

Creating avatars allows users to interact more freely and naturally in an environment. Avatars can be constructed from simple shapes such as a cube or sphere or be complex anatomically correct models of human limb segments. Creating a human-shaped avatar has already been done [Bachmann 2000]. However, people differ in size and shape from one another. Capturing individual qualities in an accurate and efficient manner remains a problem to be solved.
A system capable of automatically and accurately measuring and recording the lengths of different segments is ideal. Technologies, such as laser scanning, can capture information with great accuracy. However, these types of technologies are typically expensive. Using them will assist in the creation of a lifelike avatar. Creating a more realistic avatar could include clothing, body development (eye color, hairstyle and color, skin tone, muscle tone, etc.), and other similar factors.

5.3.4 Gait Measurements and User Position
The current system does not have any capability for measuring user movement from one position to another. A sourceless position tracking system is necessary for complete immersion in a synthetic environment. Modern technologies such as global positioning satellites (GPS) provide a sufficient solution in an outdoor setting. GPS is not perfectly accurate, however, and will not work in an indoor situation. Using a mathematical approach to measuring the length of the user’s gait combined with precision acceleration measurements provides a possible solution to this challenge.

Other technologies could be considered to be used in conjunction with the MARG sensor modules. One such system utilizes radio frequencies (RF). Radio waves can penetrate walls and the human body and are therefore able to operate without line-of-sight issues of most sourced systems. An example of an RF system currently in use is the global positioning system (GPS). GPS is best suited for outdoor situations where perfect accuracy is not required.

5.3.5 Influence of Magnetic Fields on MARG Sensor Modules
Distortions of the local magnetic field occur due to various factors. Some field variations are naturally occurring, such as a large deposit of iron ore, and some are caused by manmade structures, such as wiring in a house. Research into how these fields affect the output of the MARG sensor modules is important to determine future venues of study. If a significant effect is found, further research could be done to learn how the
magnetometers can dynamically read these effects and automatically compensate for them.

5.3.6 Mechanical Calibration of Sensor Modules
The process for calibrating the sensor modules is tedious and imprecise. It requires manual rotations at various speeds. It is nearly impossible for a human to properly calibrate the modules. The use of mechanical tilt tables and magnetic boxes to perform calibration would greatly enhance the process, as well as enhancing the output and usage of the array of sensor modules with more accurate values.

5.3.7 Limb Segment Lengths
There are current body-tracking technologies that can dynamically calculate the limb segments of the user. These systems are typically expensive. The challenge of researching how to scale an avatar to specific users would enhance the overall use of the inertial/magnetic body tracking system.
6. References


<http://java.sun.com/j2se/1.3/docs/api/>.
7. Appendices

Appendix A: Process Design Outline

I. Requirements Analysis and Definition
   A. Feasibility study
      1. Ongoing through Phase I
      2. Can the goals be met with the given resources?
   B. Define goals of system
      1. “It would be nice if the system did this”
      2. Goals are subjective and cannot necessarily be tested
   C. Requirement definition
      1. Requirements can be tested
      2. Use everyday language to define what the customer expects the system to do
      3. Basic sketches or drawings
   D. Requirement (functional or software) specification
      1. Structures system more formally
      2. Understood by customer and design team
         a. for this project, this is not an issue
         b. developer has less experience than customer
      3. Abstract description of design and implementation
      4. High-level design

II. System and Software Design
   A. Modeling techniques to be utilized
      1. Directed graphs
      2. Sketches
      3. Drawings
      4. UML
   B. Improve requirement specification
      1. Architectural design: sub-systems and relationships
      2. For each sub-system
         a. definition of its services and constraints
         b. interface with other sub-systems
         c. unambiguous, encapsulated design
         d. data structures to be used
         e. algorithms to be used
   C. Flow chart of system
      1. Bring in more details
      2. Outline interactions in more detail
   D. Hardware requirements
   E. Hardware/software interaction
   F. User interface design
      1. Style
      2. Inputs/outputs
      3. Error messages
4. Help functions

III. Implementation and Unit Testing
A. Programming methods
   1. Fault avoidance: prevent system from crashing
   2. Exception handling
B. Programming techniques
   1. Reusability
      a. inheritance
      b. composition
   2. Encapsulation
   3. Standard techniques: e.g., variables and multiWordVariables
C. Using Rational Rose to assist coding
   1. Requires good design
   2. Saves time in non-value-added processes
D. Testing units: work as stand-alone piece
   1. Boehm (1979)
      a. validation: Are we building the right product?
      b. verification: Are we building the product right?
   2. Static techniques
      a. program inspections
      b. analysis and formal verification
   3. Dynamic techniques
      a. test on the fly

IV. Integration and System Testing
A. Expensive process in terms of time
B. Bottom-up testing
   1. Lower level modules first
   2. Build to whole system
C. Combined with top-down testing
   1. Bottom-up can cause major rebuilds
   2. Requires whole system completed
D. Reliability
   1. Prioritizing various portions of system
   2. Follows up with usability
   3. Various statistics, if necessary
   4. Analysis

V. Documentation
A. Process
   1. Records process
   2. Plans, schedules, quality documents, standards
B. Product
   1. Assist end-user with use
   2. “Help” functionality
Appendix B: Relationship Diagram

NOTES:
The dashed line represents a data flow relationship in this diagram.
Every object has an ErrorLogger object that is used to write errors to a file.

ErrorLogger
## Appendix C: Requirement Specification

The software system must perform these functions.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConvertRawData</td>
<td>Converts data from raw integers to voltages</td>
<td>9 integers ranging from 0 to 32768</td>
<td>9 floats ranging from 0 to 5</td>
<td>Workstation</td>
</tr>
<tr>
<td>ConvertVoltages</td>
<td>Converts voltage values to physics measurements</td>
<td>9 floats ranging from 0 to 5</td>
<td>9 floats ranging through various values</td>
<td>Workstation</td>
</tr>
<tr>
<td>Filter</td>
<td>Uses physics measurements to create a raw quaternion</td>
<td>9 physics measurements</td>
<td>1 raw quaternion</td>
<td>Workstation</td>
</tr>
<tr>
<td>OffsetQuaternion</td>
<td>Uses fixed offset quaternion to translate raw quaternion to limb segment coordinates; entire conversion process must rate at 100 Hz</td>
<td>1 raw quaternion</td>
<td>1 offset quaternion</td>
<td>Workstation</td>
</tr>
<tr>
<td>Poll sensors</td>
<td>Wearable computer must poll and read data from sensor module through card</td>
<td>Pointer to integer array and integer primitive</td>
<td>None</td>
<td>Wearable Computer</td>
</tr>
<tr>
<td>Relay raw data</td>
<td>Wearable computer must transfer data over network connection to workstation</td>
<td>Pointer to integer array and integer primitive</td>
<td>UDP packet of data (see inputs)</td>
<td>Wearable Computer</td>
</tr>
</tbody>
</table>
Function Name: Archive sensor calibration data  
Description: Once calibrated, the data for calibration should be saved to a file and loaded at the program’s onset  
Inputs: Calibration data  
Outputs: File  
Location: Workstation

Function Name: Archive sensor data  
Description: As the data goes through the conversion process, the data should be able to be saved at any stage  
Inputs: Data at some point through conversion  
Outputs: .dat file  
Location: Workstation

Function Name: Retrieve UDP data  
Description: Workstation must be able to receive packets of data from the wearable computer using a UDP socket  
Inputs: None  
Outputs: Data from wearable computer  
Location: Workstation

Function Name: Multicast data  
Description: After the data has been converted, it should be multicast on a known address so that it can be picked up by any display units  
Inputs: Offset quaternions  
Outputs: Multicast data packet  
Location: Workstation

Function Name: Post-processing data  
Description: After data is archived, the system must be able to re-read and properly convert the data for replay or testing. The data should also be able to be read by any program  
Inputs: Data from file  
Outputs: 9 integers or 9 floats or 1 quaternion  
Location: Workstation

Data flow will begin from the sensor modules, which produce an array of integers. This data is read by the wearable computer. The wearable computer will send the data via datagram packet to the workstation. The workstation will convert the data into quaternions and send the quaternions over a multicast connection. The workstation will also have an intuitive user interface. Other functionalities of the workstation are listed...
above. Figure A shows one instance of how data flows through the system. The figures at the top represent objects in the system. The dashed lines provide a simple grid, and the solid lines running horizontally represents where the data is flowing and how it is flowing.