NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

APPLICATION OF INERTIAL SENSORS AND FLUX-GATE MAGNETOMETER TO REAL-TIME HUMAN BODY MOTION CAPTURE

by

William Frey III

September 1996

Thesis Advisors:	Robert McGhee
	Michael Zyda
Second Reader:	Russ Whalen

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE September 1996	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. APPLICATION OF INERTIAL SENSORS AND FLUX-GATE MAGNETOMETER TO REAL-TIME HUMAN BODY MOTION CAPTURE			5. FUNDING NUMBERS
6. AUTHOR(S) William H. Frey III			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)			10.SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Human body tracking for synthetic environment interface has become a significant human-computer interface challenge. There are several different types of motion capture systems currently available. Inherent problems, most resulting from the use of artificially-generated source signals, plague these systems. A proposed motion capture system is being developed at the Naval Postgraduate School which utilizes a combination of inertial sensors to overcome these difficulties. However, the current design exhibits azimuth drift errors resulting from the use of inertial sensors.

This thesis proposes a new method of compensating for azimuth drift using a three-axis fluxgate magnetometer. The fluxgate magnetometer is capable of azimuth drift compensation since its sensitive axis is not collinear with the local vertical. This thesis includes a program for simulating the operation of a fluxgate magnetometer in C++. The included C++ code simulates a fluxgate magnetometer and provides an estimate of azimuth based on the magnetometer's output which is typically within five degrees of the actual azimuth. Real magnetometer data for testing and verification was accomplished by bench testing a real fluxgate magnetometer.

14. SUBJECT TERMS Human Body Tracking, Inertial, Magnetometer			15. NUMBER OF PAGES 170
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18 298-102

APPLICATION OF INERTIAL SENSORS AND FLUX-GATE MAGNETOMETER TO REAL-TIME HUMAN BODY MOTION CAPTURE

William Frey
Lieutenant, United States Navy
B.S., Oregon State University, 1989

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1996

Author: [illegible]
William Frey III

Approved by: Robert McGhee
Robert McGhee, Thesis Advisor

Michael J Zyda
Michael Zyda, Thesis Advisor

Russell H Whalen
Russ Whalen, Second Reader

Ted Lewis
Dr. Ted Lewis, Chairman
Department of Computer Science

ABSTRACT

Human body tracking for synthetic environment interface has become a significant human-computer interface challenge. There are several different types of motion capture systems currently available. Inherent problems, most resulting from the use of artificially-generated source signals, plague these systems. A proposed motion capture system is being developed at the Naval Postgraduate School which utilizes a combination of inertial sensors to overcome these difficulties. However, the current design exhibits azimuth drift errors resulting from the use of inertial sensors.

This thesis proposes a new method of compensating for azimuth drift using a three-axis fluxgate magnetometer. The fluxgate magnetometer is capable of azimuth drift compensation since its sensitive axis is not collinear with the local vertical. This thesis includes a program for simulating the operation of a fluxgate magnetometer in C++. The included C++ code simulates a fluxgate magnetometer and provides an estimate of azimuth based on the magnetometer's output which is typically within five degrees of the actual azimuth. Real magnetometer data for testing and verification was accomplished by bench testing a real fluxgate magnetometer.

TABLE OF CONTENTS

ACKOWLEDGMENT

The author wishes to acknowledge the very significant contributions and efforts of the following individuals:

Doctor Michael Zyda, for his implicit trust in my ability to logistically coordinate and produce the 1994 Interactive Three-Dimensional Graphics conference in Monterey, California, and for giving me the opportunity to write a technical report, *Off-the-Shelf, Real-Time, Human Body Motion Capture for Synthetic Environments* , which launched me into my thesis with both feet.

Doctor Robert McGhee, for his understanding of the trials and tribulations of a Master's candidate in the process of writing a thesis, and for his untiring tutelage and support which made it possible for me to finish. Doctor McGhee is a professor who truly loves to teach. His efforts in and out of class were absolutely invaluable.

Doctor Jim Clynch, for his generous loan of the equipment used in magnetometer bench testing and for his technical support and advice.

Russ Whalen, for his superb technical support during magnetometer bench testing, for his advice during the entire course of this thesis and for his helpful comments in his capacity as my second reader.

Shawna McCartney, for her patience, understanding and support which has gotten me through the final stages of this thesis. Without her loving support and encouragement, I would never have made it to the end.

My parents, Jerry and Judith Harper, whose words of encouragement through trying times made it possible for me to overcome the obstacles which lay before me.

My peers in Naval Postgraduate School, class CS-51, for their constant companionship and support. Misery truly loves company.

Thank you, all!

I. INTRODUCTION

A. RATIONALE FOR HUMAN BODY TRACKING

For a number of years, researchers have been attempting to create believable three-dimensional worlds inside a computer for a variety of purposes including data visualization, computer-aided design, training of all sorts, the control of remote robots and manipulators (tele-operation), artificial enhancement of the real world, and entertainment. These researchers, for the most part, have failed to hit the 'total immersion' mark, although consumers are generally willing to overlook inadequacies while exploring new technological advances. [NRC95]

There are various reasons why synthetic environments have failed to reach the goal of 'total immersion'. One of the main reasons is the lack of a natural interface between the computer and the human machine. One might say that people have grown quite accustomed to using a keyboard and a mouse to communicate with their computer, and indeed, some people are very adept at operating a computer using these devices. However, keyboards and mice are not present when people exit their domiciles and interact with the real world. People use all of their senses (the five basic: sight, hearing, touch, taste and smell) to receive information about the world they live in and they use their body motions to act on objects in that world.

The fact that people have so many senses to sample the world's information stream and that they use their entire bodies to interact with that world is one of the main reasons that synthetic environments fall short. The goal of virtual environments, in general, is to fool the human sensor suite enough to make the human participant think that he or she is interacting with a real environment. While fooling the human visual and auditory senses has become fairly routine for synthetic environment researchers, fooling the other human senses has been found to be very difficult. There is much work left to be done on the human-computer interface. [NRC95]

The issue of allowing people to interact naturally with a synthetic environment has been the subject of much debate. One thing is clear: If the user is to interact with a synthetic environment in a way which is perceived to be natural, then an interface device must be provided which is capable of determining what the user is doing without interfering with their motion or encumbering their body. This device must accurately capture the user's motions and supply them to the synthetic environment generator with an update rate sufficient to provide the user with real-time response to their actions.

B. ORGANIZATION OF THESIS

This thesis focuses on the issues surrounding the development a human body tracking system which utilizes inertial and magnetic sensors to overcome some of the drawbacks of motion capture systems. Chapter II discusses the fundamentals of human body motion capture for synthetic environments. Chapter III covers the fundamentals of inertial sensing and Chapter IV details the application of inertial sensors to human body motion capture. Chapter V discusses the use of a three-axis flux-gate magnetometer for azimuth drift compensation of a device which utilizes gravitational sensors (accelerometers) for orientation determination. Also covered in Chapter V is the development of a C++ simulation of a three-axis flux-gate magnetometer and its use in azimuth drift compensation. Chapter VI discusses the adequacy of orientation-only tracking for human body motion capture for synthetic environments. Chapter VII summarizes this thesis and addresses topics for future work in the area of human body motion capture using inertial sensors.

II. MOTION CAPTURE FOR SYNTHETIC ENVIRONMENTS

A. HUMAN BODY MOTION CAPTURE

Figure 1 shows the configuration of human body parts which must be tracked for the application envisioned in this thesis. In general, for the degree of realism envisioned in this thesis, if one desires to track the entire human body, there are fifteen major parts to track independently. The major portions of the body that must be tracked are the head (normally tracked as an input to the graphics rendering software used to drive a head-mounted display), torso-clavicle region, abdomen-hips region, upper legs, lower legs, feet, upper arms, lower arms and hands.

While more body parts than those shown in Figure 1 may be tracked, tracking any more than these may result in diminished returns. For instance, it may not be worth the extra equipment required to track the user's back and shoulders as several separate entities. This is, of course, dependent upon the user's application. If the user's application requires separate tracking of the shoulders and back or separate tracking of the parts of the foot, then the motion capture system must be capable of adapting to these needs. In addition, tracking more parts than required necessitates further encumbering the user, which may detract from the success of the virtual environment interface.

There is also a trade-off between the time required to process the physical information from the parts of the body being tracked and the time required to calculate the positions of the parts of the body that are not being tracked. For every body part that is not tracked, the system must estimate its orientation using inverse kinematics. Inverse kinematics algorithms are typically computationally complex and it is the author's opinion that they require more computing power than measuring the orientation of the parts directly. For instance, direct tracking of only the shoulder and hand positions requires that the position of the elbow and the joint angles

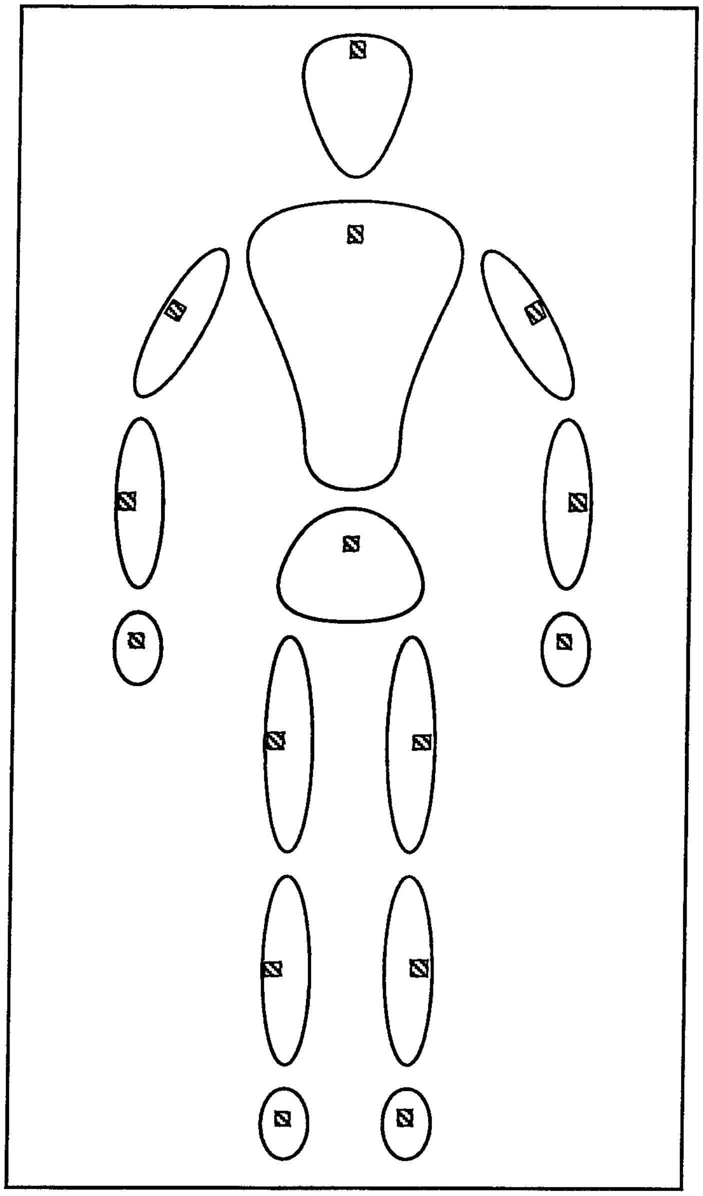


Figure 1 -- Proposed Human Body Tracking Configuration

of the shoulder, elbow and wrist be estimated. This is possible, but not as physically accurate or computationally efficient as tracking the parts directly. [WALD95]

Tracking the body parts directly also has its drawbacks. First, placing sensors on the body encumbers the user. One of the goals of a synthetic environment is to completely immerse the user in a believable world where it will seem natural to be there. Placing more sensors than necessary on the user's body is contrary to this goal. In addition, the cost of a body-tracking system varies almost linearly with the number of body parts tracked; more sensors equates to higher cost.

The information required from each body part tracker varies. There are some devices which provide all 6 degrees of freedom (DOF) (spatial position and orientation) for each tracked object. This is overkill for tracking the human body (or any other articulated body). For the fifteen body parts displayed in Figure1, it can be shown that spatial position is required for only one (base) body part. For all other body parts, 3 DOF (orientation only) is sufficient to completely describe the pose of the entire human body (or any other articulated body). All parts other than the base part are spatially positioned relative to the base part. This articulation technique is described in detail in Chapter VI.

Using this scheme and the configuration of Figure 1 results in a system that can adequately track the human body using only 48 DOF (NOTE: degrees of freedom are "...independent position variables which would have to be specified in order to locate all parts of the mechanism." [CRAI89] For the purposes of this thesis, 6 DOF refers to the following variables: The earth-fixed orientation angles -- azimuth, elevation and roll -- and the earth-fixed spatial position variables -- x, y and z.).

The most popular method for measurement of human body part position and orientation directly involves the use of electromagnetic fields. An electromagnetic field is generated by a stationary transmitter and is detected by multiple receivers mounted on the user's body. One receiver is attached to each of the user's body parts. It is used to detect the spatial position and orientation of the body part relative to the stationary transmitter. This system will be described in more detail later in this chapter.

Some systems, usually mechanical in nature, track body joint angles rather than body part positions and orientations. While this method leads to a direct and reliable means of tracking the human body, it is typically very encumbering to the user. Some of the systems that use this method are exo-skeletal (attached to the body to measure joint angles directly). These systems are naturally sensitive to the differences between users' bodies and are reliant on the size of the user's body parts (upper arm, forearm, etc.) to determine the spatial positions of the user's extremities. Thus they must be re-calibrated for each new user to ensure proper operation. Of course, it is possible to store a user's body part dimensions for later use once they have been measured.

There are various means of capturing object position and orientation which are currently employed in the field of synthetic environments. Each has weaknesses which make it unsuitable, or strengths which make it particularly suitable, for certain applications. The next section focuses on the various means of human body motion capture, their technological capabilities, advantages and disadvantages.

B. CURRENT METHODS OF MOTION CAPTURE

This section details the main methods of human body motion capture currently available. The majority of the information in this section comes directly from Virtual Reality: Scientific and Technological Challenges [NRC95]. Additional information was obtained directly from the equipment manufacturers themselves and can be found in [FREY95].

1. Mechanical Systems

Mechanical position tracking devices can be separated into body-based (exo-skeletal systems) and ground-based systems. Body-based systems are those which are mounted on, or carried on, the body of the user and are used to sense either the relative positions of various parts of the user's body or the position of an instrument relative to a fixed point on the user's body. Ground based systems are typically not carried by the user but are mounted on some fixed surface (i.e. the user's desk or the floor) and are used to sense the position of an implement relative to that fixed surface.

Body-based systems are typically used to determine either the user's joint angles for reproduction of their body in the synthetic environment, or to determine the position of an endeffector (the user's hand, foot, etc.) relative to some point on the user's body. Since the body-based systems are used to determine the relative position between two of the user's body parts, the devices must somehow be attached to the user's body. This particular issue has raised many questions: How is the device attached to the body in a way which will minimize relative motion between the attachment and the soft body part it is being attached to? How are the joints of the device aligned with the user's joints to minimize the difference in the centers of rotation (a significant source of error)?

Some other problems associated with body-based tracking systems are specifically caused by the device being attached to the user's body. These systems are typically very obtrusive and encumbering. They do not allow the user complete freedom of movement and they detract from the possibility of the user experiencing complete immersion into the synthetic environment. Body-based mechanical systems are, however, quite accurate and do not experience problems such as measurement drift (the tendency of the device's output to change over time with no change in the sensed quantity), interference from external electromagnetic signals or metallic devices in the vicinity, or shadowing (loss of sight of the tracked object due to physical interference of another object). [NRC95]

Ground-based systems are typically used to determine the position and orientation (6 DOF) of an implement manipulated by the user relative to some fixed point not on the user's body. These devices are not typically attached to the user's body, provided the user can grasp the manipulator in a rigid manner. Like body-based mechanical systems, they are typically very accurate and are not plagued by measurement drift errors, interference or shadowing.

Ground-based systems do suffer from one thing which the body-based systems do not: They confine the user to work within the space allowed by the device. Usually this means that the user is confined to work in a space the size of a large desk. If the application does not require the user to move around much throughout the task (i.e. the user remains seated), this is not usually considered a problem. [NRC95]

Mechanical tracking systems are the best choices for force-feedback (haptic) devices since they are rigidly mounted to either the user or a fixed object. Haptic devices are used to allow the user a "sense of touch". The user can feel surfaces in the synthetic environment or feel the weight of an object. The device can apply forces to the user's body so that the user can experience a sense of exertion. Ground-based systems are typically the best choice for incorporation of haptic devices due to their rigid mounting on a fixed surface. [NRC95]

Mechanical tracking systems also typically have low latencies (the time required to receive useful information about a sensed quantity) and high update rates (the rate at which the system can provide useful information) [NRC95].

2. Electromagnetic Systems

Electromagnetic tracking systems are currently the most widely used systems for human body tracking applications. They employ the use of artificially-generated electromagnetic fields to induce voltages in detectors attached to the tracked object. Three orthogonal electromagnetic fields are generated by a stationary transmitter. These fields interact with the three orthogonal coils in each detector attached to the tracked object. Induced voltages are generated in the detector coils which are proportional to the spatial orientation of the detector relative to the transmitter.

These tracking systems are fairly inexpensive and can be used to track numerous objects (body parts) with acceptable position and orientation accuracies (typically advertised to be on the order of 0.1 inches and 0.5 degrees). They do not suffer from shadowing effects, but are typically plagued by a sensitivity to background magnetic fields and interference caused by metal devices in the vicinity. Since they are reliant on the magnetic fields generated by the transmitter, these systems are called "sourced" systems and have a limited tracking area, typically the size of a small room. [NRC 95]

Electromagnetic tracking systems can employ either AC or DC magnetic fields. Those employing DC magnetic fields are typically less sensitive to interference caused by metallic objects in their vicinity [NRC95].

3. Acoustic Systems

Acoustic tracking systems utilize high frequency sound waves to track objects by either the triangulation of several receivers (time-of-flight method) or by measuring the signal's phase difference between transmitter and receiver (phase-coherence method).

The "time-of-flight" method of acoustic tracking uses the speed of sound through air to calculate the distance between the transmitter of an acoustic pulse and the receiver of that pulse. The use of one transmitter on a tracked object and a minimum of three receivers at stationary positions in the vicinity allow an acoustic system to determine the relative position (3 DOF) of the object via triangulation. This method limits the number of objects tracked by the system to one. An alternative method has been devised in which several transmitters are mounted at stationary positions in the room and each object being tracked is fitted with a receiver. Using this method, the positions of numerous objects may be determined simultaneously. [NRC95]

Note that the use of one transmitter (or one receiver) attached to an object can resolve only position (3 DOF). Three transmitter (receiver) sets mounted on the same object can be used to determine the position *and* orientation (6 DOF) of the object. The desire to track more than just the position of an object suggests that the second method (multiple stationary transmitters with body-mounted receivers) may be preferable.

The other method of acoustic tracking, phase-coherent tracking, may be used to achieve better accuracies than the time-of-flight method. The system does this by sensing the signal phase difference between the signal sent by the transmitter and that detected by the receiver. If the object being tracked moves farther than one-half of the signal wavelength in any direction during the period of one update, errors will result in the position determination. Since phase-coherent tracking is an incremental form of position determination, small errors in position determination will result in larger errors over time (drift errors). [NRC95]

Some problems associated with both acoustic tracking methods result from the line-of-sight required between transmitter and receiver. This line of sight requirement obviously plagues the devices with shadowing problems. It also limits their effective tracking range, although they typically have better tracking ranges than electromagnetic systems. Unlike electromagnetic systems, they do not suffer from metallic interference, but they are susceptible to interference caused by reflections of the acoustic signals from hard surfaces and interference from ambient noise sources.

4. Image-Based Systems

Image-based systems are lumped into two broad categories; those that use active targets and those that use passive targets (or no targets). Targets are devices which, when placed on the object to be tracked, are visible to the tracking system. In both systems, cameras are used to record the object being tracked and detect the motion of the targets on the object. Typically, multiple cameras are used so that the object may be tracked in three dimensions instead of just two. While only two cameras are required to achieve three dimensional tracking, more are typically used to provide redundancy in an effort to prevent shadowing of the targets.

Image-based systems which use targets attached to the object being tracked are called marker systems. The targets used in active marker systems are typically infrared light-emitting diodes (IRED's) which emit light visible to the system but not to the user. As in acoustic systems, the detectors, or cameras, may be placed either on the tracked object or at stationary points around the object. Obviously, cameras placed on a human body would be more obtrusive. For this reason, placing the targets on the body and the cameras at stationary points in the room is normally preferable.

Each camera is placed so that it has a unique perspective of the targets. Thus triangulation of the targets can be used to track them in three dimensions. This technique reveals the first major problem with image-based systems; determining correspondence of targets in each of the camera views. In order to use several views of the same target to triangulate its position, a target must be distinguishable from the other targets around it. One method of distinguishing the targets is to pulse their outputs in sequence with camera detection. Once the targets can be distinguished, the remaining question is, how many may be used simultaneously. If orientation of the object is desired in addition to position, at least three targets must be placed on the same object and their differences in position used to determine the orientation of the object.

Image based systems rely on the cameras being able to detect the targets at any given instant in time. If an object passes between a marker and a camera during the detection interval, the camera will fail to detect the marker. If this condition persists for a long enough period of time, tracking of the object will fail. Failure in tracking a human body may be caused by as simple a thing as one body part obscuring another from all of the camera viewpoints. This effect is called shadowing. As mentioned before, shadowing may be minimized by the use of multiple, redundant cameras, but it cannot be totally eliminated.

As would be expected, multiple-source image processing requires a level of computational complexity not required by the other methods of motion capture. The combination of the computational requirements and the use of multiple high-resolution cameras makes image-based tracking one of the most expensive body tracking solutions available.

While they are not yet feasible for accurate body tracking in synthetic environments, lower-cost, image-based systems would be very suitable for gesture recognition systems. For example, the detection of hand or arm signals from the user directing the computer on which way the user wants to travel in the environment.

5. Optical Systems

There are numerous means of optical tracking, each employing a slightly different technique with differing equipment. This section will give a short synopsis of the most prevalent methods followed by the pros and cons of optical tracking methods in general.

a. Position-Sensing Detector (PSD) Systems:

Position-sensing detectors (PSD's) are photo-electronic devices, each made from a slice of silicon doped with materials which form a PN junction. The PN junction is light sensitive and incident light will cause it to generate an electrical current. This electrical current is inversely proportional to the distance between the image of the incident light source and the sensing electrode. When a light source is positioned over the device, its location in the x-y coordinates of the PSD may be determined by comparing the relative strengths of current signals from various attached electrodes.

When several of the PSD's are utilized from various positions, triangulation using the signals from the devices may be used to determine the 3 DOF position of a light source. This method is very similar to the image-based tracking method described above, the difference being the sensing device. It suffers from all of the same problems which afflict image-based tracking systems.

b. Structured Light Systems:

Typically in structured light systems, a laser beam and beam-forming optics are used to create a known pattern of coherent light which is then scanned across the scene. A camera is used to capture the scene as the light is scanned across it. The intersection of the camera plane and the laser light beam reflecting from the surfaces of the scene creates a three-dimensional coordinate system [NRC95].

c. Laser Radar:

The concept of laser radar is similar to that of time-of-flight acoustic systems. A laser is used to scan an object and the returning, reflected laser light is detected. The difference in time between sending the beam and receiving the reflected light is a function of the range to the reflecting surface. If the beam is scanned over a scene, a three-dimensional picture of the scene is generated [NRC95].

d. Laser Interferometry:

This system uses a steered laser beam to track a retro-reflector mounted on the object being tracked. The angle subtended by the steered laser beam, in two dimensions, and the time-of-flight of the laser light forms a three-dimensional space. Another method uses several lasers, each tracking a retro-reflector from a unique perspective, to form the three-dimensional space [NRC95].

In all of the structured light tracking methods, the use of laser light tends to make the system extremely accurate. However, none of the above systems is capable of tracking more than a few objects simultaneously, and all are susceptible to shadowing. These problems tend to make purely optical tracking methods insufficient for real-time tracking of the entire human body.

C. OTHER EXPERIMENTAL MOTION CAPTURE METHODS

1. Spread-Spectrum

Another method of navigational position determination is the use of the Global Positioning System. This technique uses a constellation of satellites orbiting the earth which emit signals intercepted by a navigational receiver. Each signal is decoded by the receiver to determine the exact distance between the satellite and the receiver. Triangulation is then used to determine the receiver's spatial position.

This technology can be adapted to provide a very accurate position determination on a much smaller scale. The construction of a large area (like a football stadium) containing a set of low strength spread spectrum transmitters would allow a suitably instrumented human body the freedom to roam around, while each GPS-style receiver attached to the user's body would determine the position of each limb. The use of three receivers attached to each limb would allow the determination of limb orientation as well as position.

If, as with the proposed inertial tracking system, the positional data was transmitted from the user's body to the computer system via wireless communication means, the user would be entirely untethered. Thus the user would be free to roam anywhere within the effective range of the spread-spectrum transmitters.

The primary draw-backs of spread-spectrum human body tracking would be range restrictions, the potential for high-frequency electromagnetic radiation exposure of the user, and multi-path signal reception in confined spaces. In addition, the accuracy achievable by affordable spread-spectrum systems is yet to be determined and the initial cost of this type of system is likely to be quite high. As research continues and the technology is made more widely available, the price should fall [BIBL94].

D. SUMMARY

Real-time, human body motion capture can be used as an outstanding human / computer interface paradigm for synthetic environments and there are several methods available to accomplish this. However, all of the current motion capture systems have characteristics which either make them complicated or unsuitable for use as a synthetic environment interface. The following chapters detail an inertial human body motion capture system that has all of the best characteristics of motion capture systems to date and few of their drawbacks.

III. FUNDAMENTALS OF INERTIAL SENSING

A. HISTORY OF INERTIAL SENSING

In the early days of ship navigation across large distances, various techniques were employed to determine the position of the ship on the Earth. Navigators observed that stars were a valuable reference for position and heading as they always seemed to be in the same relative position at a certain time of night. This, however, was only good if the navigator could see the stars and possessed an accurate timepiece. When there was significant cloud cover, they realized that they required some other method of position determination.

Near land, navigators used a technique called "piloting". This relied on the use of landmarks as references of the ship's position. Again, this method required that the navigator be able to see the land which he was referencing. If the distance from land was too great (on the order of 30 miles or more) or there was a significant amount of atmospheric disturbance (like fog), then the navigator's system of piloting could not be performed.

What the navigators found they desired was some sort of self-contained navigation system that could be relied upon when neither land nor the stars were visible. Very early on, navigators realized that they could use a combination of the ship's speed, the ship's heading and an accurate measure of time to estimate their position based on a known earlier position. This method was called "dead-reckoning". [BOWD77]

Dead-reckoning required a fairly accurate heading reference so that the ship's direction of travel was known. Typically, a standard ship's compass was found to be sufficient.

The ship's speed was determined using a log line or a towed log. A log line was a length of rope with markers that was thrown over the side of the ship and allowed to drift from the bow of the ship past the stern. Since the length of the ship was known, the measured time that it took for the markers to drift from one end of the ship to the other was inversely proportional to the ship's speed. The towed log was a line with a propeller-type device on the end. This line was towed behind the ship and its rate of rotation observed. The rate of rotation was proportional to the ship's speed.

The remaining requirement was an accurate chronometer. The ship's navigator typically possessed a very accurate time-piece with which to measure the time difference desired.

With these three pieces of information, the navigator was able to determine the ship's approximate position by multiplying the ship's speed by the time difference, taking into account the ship's direction of travel, and adding the result to the ship's initial position. This geometric solution worked fairly well for short periods of time, but was very tedious and tended to rely entirely on the discipline of the navigator for careful calculation. In addition, many sources of errors were found to exist, including wind and ocean current effects which the navigator could not accurately account for. Thus, yet another method of self-contained navigation was desired that would minimize the effects of dead-reckoning and allow the ship's position to be determined without any external sources of information.

In his study of the physical characteristics of our environment over 300 years ago, Sir Isaac Newton established the principles of inertial navigation. His laws of mechanics established the foundation for self-contained inertial navigation systems. Nevertheless, it took nearly 300 years for the technological base to sufficiently develop before Newton's principles could be put to practical use in the first self-contained inertial navigation system. [ODON64]

> Inertial navigation systems, self-contained, independent of electromagnetic radiation and the Earth's magnetic field, are the contribution of modern technology to progress in dead-reckoning navigation. These systems require no wind or ocean-current data, no detectable radiation, no magnetic compass, no time-shared usage of ground facilities, no operator time during flight, and no special maps. Their accuracy, independent of operating altitude and terrain, is limited almost solely by the accuracies of their component instruments. [ODON64]

While O'Donnell was discussing inertial navigation as it applies to aviation, the principles are the same for ship navigation and inertial-referenced position determination in general.

Standard inertial navigation systems have been employed for a very long time on Naval ships, aircraft, missiles and other bodies which needed to know where they were in an inertial reference frame. They have relied upon being able to detect minute changes in a body's linear acceleration, and then applying that acceleration, doubly integrated in time, to a body's initial position and velocity to determine that position of the body at some later point in time. The standard technique of double integrating linear acceleration is also referred to as "dead-reckoning", although this usage is misleading since the term originally referred to the single integration of velocities.

B. UTILIZATION OF INERTIAL SENSORS

In order to determine a body's position in time, relative to an inertial reference frame, the following information must be known about the body:

1) The body's initial position and velocity,
2) The body's orientation relative to the reference coordinate system as a function of time,
3) The body's linear acceleration vector as a function of time and
4) The time difference between the initial time and the time in question.

From these quantities, the body's current position may be determined by the double integration of the time-variant linear acceleration over the path of travel of the body, taking into account its orientation relative to the inertial reference frame.

A standard inertial navigation system typically employs some type of tilt or angle measuring platform to measure the orientation of the body relative to the reference coordinate system. The typical orientation vector consists of azimuth (the body's heading or rotation about the inertial reference frame's vertical axis), elevation (the body's pitch or rotation about its lateral axis) and roll (the body's rotation about its longitudinal axis). These three angles serve to uniquely define the orientation of the body relative to the inertial reference frame.

The standard "strap-down" inertial navigation system employs linear acceleration sensors, which provide the body's instantaneous linear acceleration in its own coordinate system (the body-fixed coordinate system), which is then converted into the equivalent linear acceleration in the earth-fixed coordinate system by the application of transformations based on the body's orientation relative to the inertial reference frame. Non-strap-down systems use the orientation sensors and servo-motors to maintain the linear accelerometer platform (or stable table) at some zero reference orientation relative to the inertial reference frame. This ensures that the directions of the sensed linear accelerations remain constant with respect to the inertial reference frame so that no transformations of the acceleration data are required prior to their use in the body's position estimation. [ODON64]

While the methods described above may seem like an ideal way of maintaining a position estimate of a body, they are in fact plagued by sensor measurement errors which are inherent in the devices used. These errors may be partially compensated for but never eliminated, resulting in the constant search for better quality inertial instrumentation.

The first of these errors is called "drift", or the tendency of bias errors in the angular rate sensors of the inertial platform to cause ever-increasing orientation measurement errors. These errors result from the single integration of the bias-ridden angular rate signal. This integration allows a steady build-up of error over time, which results in an incorrect estimation of the orientation of the body relative to the earth-fixed coordinate system and a corresponding error in the body's position estimate. If the bias of the angular rate sensors was a constant, then compensation would be simple. However, the sensor bias typically changes over time in an unpredictable manner, so no complete compensation is possible.

If another method of determining instantaneous orientation exists, drift may be compensated for by a periodic adjustment of the inertial sensor suite orientation to this external reference. In standard inertial navigation techniques, this is called a "fix". By taking a fix, the build-up of bias errors is periodically returned to zero. By keeping the length of time between fixes, the "fix interval", below a certain specified length of time, the bias errors can be made to be relatively insignificant. If however, the fix interval is not strictly adhered to, then the bias errors will push the position estimate out of tolerance and, in the world of standard inertial navigation, the ship will have the potential to run aground. [BOWD77]

The amount of drift, or bias, present is a characteristic of the angular rate sensors themselves. Typically, the higher the angular rate sensor quality, the lower the bias error. The lower bias error means that the fix interval may be longer. In other words, a fix is not required as often, meaning less time spent taking navigational fixes, and less potential for navigational errors.

Linear acceleration sensors also are plagued by bias errors and thus, also suffer from drift. Their errors, however, are compounded by the fact that, the desired position data must be obtained by double integration of the linear acceleration measurements. This causes an error in the position estimate proportional to time-squared, rather than just time. This error may also be compensated for by periodic "fixes" of the ship's actual position but, given the same sensor quality, the fix interval will be much shorter than that required for the angular rate sensor bias compensation alone.

C. A DIFFERENT METHOD OF USING INERTIAL SENSORS

The above description of inertial navigation centers on position estimation using inertial sensors. While determination of position can be useful in tracking the human body, it is far more useful to be able to determine the spatial orientation of the individual body parts (as discussed in Chapter II). Thus, a more appropriate use of linear accelerometers is as an attitude reference. This is accomplished by attaching three orthogonal linear accelerometers to an object moving at constant velocity, in an earth-fixed inertial reference frame and measuring their outputs. Since linear accelerometers are sensitive to gravity as well as linear accelerations, in a system which is not continuously accelerating, the three linear accelerometers will produce an output vector indicating the direction of the local vertical. The local vertical vector can be used to determine a stationary object's pitch and roll relative to the Earth-fixed inertial reference frame.

However, objects in inertial reference frames seldom remain at constant velocity, and linear accelerometers are sensitive to the forced accelerations of the object they are attached to as well as gravity. This characteristic of linear accelerometers is sometimes referred to as "slosh". [FOXL94] Slosh prevents a set of linear accelerometers, alone, from giving a reliable indication of an accelerating object's pitch and roll relative to the local vertical. Specifically,

> It can be shown that in a Mach 1 vehicle executing a turn with a radius of 1,000 miles, this error in indicated vertical becomes approximately 5 millirad (although at a velocity of 180 knots the error would be only 0.5 millirad). For a star-shot attitude reference for precise navigation, this error, amounting to some 20 miles for the Mach 1 case, is much too large, so some other method of determining local level must be used. [ODON64]

Note, however, that the accelerometers, when being used to indicate the local vertical, are not subject to the error build-up caused by the double integration of their bias errors as before. Instead the errors show up as random local vertical indication errors which can be minimized by appropriate filtering. [BACH96] Thus, if some method is used to compensate for the "slosh" of linear accelerometers, the pitch and roll of a body, relative to an Earth-fixed inertial reference frame, may be reliably determined.

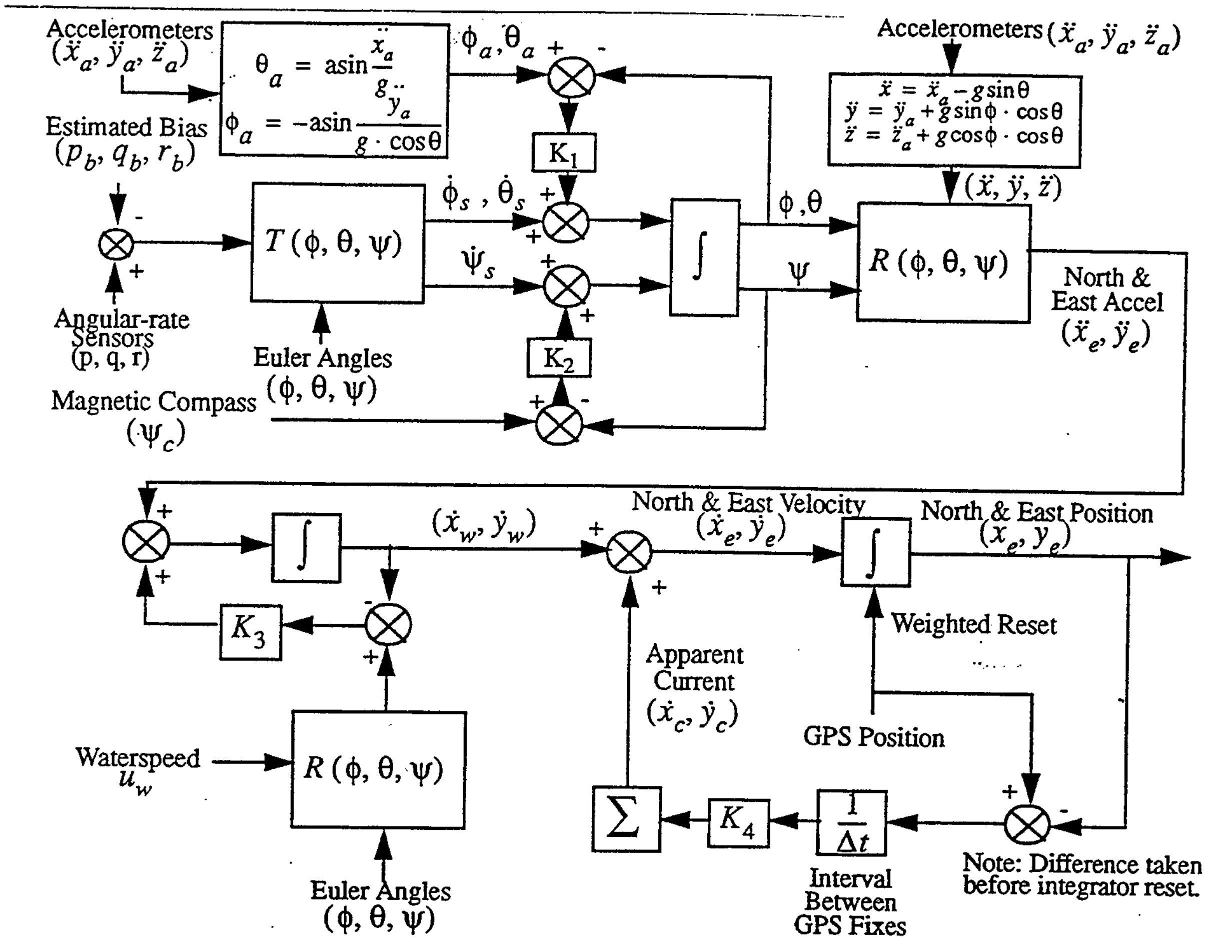
Also note that, if a body is not continuously accelerating in one direction (a characteristic of almost all real objects) then the average of the object's forced linear acceleration vector will eventually be zero, leaving only the gravity vector indicated by the output of the accelerometers. In other words, the relative long-term average of the accelerometer outputs yields the gravity vector, which can then be used as an orientation reference in an Earth-fixed coordinate system. However, as stated, this is only a long term solution and must still be compensated for slosh for short term estimates of orientation.

One choice of compensation that immediately comes to mind is a combination of linear accelerometers and angular rate sensors, since they are the two main inertial sensors available, and both can be used to estimate angular orientation. As previously discussed, the bias errors of angular rate sensors render them , at best, a short-term solution for the determination of angular orientation. Thus, it can be seen that, if some method is devised to combine the high-frequency (short term) characteristics of the angular rate sensors with the low-frequency (long term) characteristics of the linear accelerometers, a stable indication of angular orientation may be generated.

It would seem natural to apply a low-pass filter to the linear accelerometer outputs and a high-pass filter to the integrals of the angular rate sensor outputs and combine the results to obtain the desired orientation vector. The problem then becomes one of cutoff frequency selection for the low- and high-pass filters, and what method of combination to use for the filter outputs.

McGhee, et. al., have been experimenting with an inertial navigation system for autonomous underwater vehicle (AUV) control that utilizes a combination of angular rate sensors, linear accelerometers and a flux-gate compass to estimate (by dead-reckoning) the Earth-fixed position of the AUV between position fixes from an onboard GPS receiver [MCGH95] [BACH96]. These works describe a method of using angular rate sensors to compensate for the slosh of linear accelerometers, while estimating angular orientation. In addition, a flux-gate compass is used in combination with the angular rate sensors to estimate the AUV heading.

The strap-down inertial navigation approach, as shown in Figure 2, involves the use of complementary filtering of the input signals in which the low frequency characteristics of the linear accelerometer and flux gate compass outputs and the high frequency characteristics of the angular rate sensor outputs are combined to produce a stable orientation vector of the AUV relative to the Earth-fixed inertial reference frame. This method relies on the fact that linear accelerometers are sensitive to gravitational acceleration as well as forced linear accelerations and that the flux gate compass is sensitive to the Earth's magnetic field. Either sensor by itself has singularities, or sensor orientations where the indicated orientation is not unique, or where there are an infinite number of orientation solutions based on the sensor outputs.



Accelerometers $(\ddot{x}_a, \ddot{y}_a, \ddot{z}_a)$
$\theta_a = \mathrm{asin}\frac{\ddot{x}_a}{g}$
$\phi_a = -\mathrm{asin}\frac{\ddot{y}_a}{g \cdot \cos\theta}$
Estimated Bias (p_b, q_b, r_b)
ϕ_a, θ_a
K_1
$\dot{\phi}_s, \dot{\theta}_s$
$\dot{\psi}_s$
$T(\phi, \theta, \psi)$
Angular-rate Sensors (p, q, r)
Euler Angles (ϕ, θ, ψ)
K_2
Magnetic Compass (ψ_c)
ϕ, θ
ψ
Accelerometers $(\ddot{x}_a, \ddot{y}_a, \ddot{z}_a)$
$\ddot{x} = \ddot{x}_a - g\sin\theta$
$\ddot{y} = \ddot{y}_a + g\sin\phi \cdot \cos\theta$
$\ddot{z} = \ddot{z}_a + g\cos\phi \cdot \cos\theta$
$(\ddot{x}, \ddot{y}, \ddot{z})$
$R(\phi, \theta, \psi)$
North & East Accel $(\ddot{x}_e, \ddot{y}_e)$
$(\dot{x}_w, \dot{y}_w)$
North & East Velocity $(\dot{x}_e, \dot{y}_e)$
North & East Position (x_e, y_e)
K_3
Apparent Current $(\dot{x}_c, \dot{y}_c)$
Weighted Reset
GPS Position
Waterspeed u_w
$R(\phi, \theta, \psi)$
Euler Angles (ϕ, θ, ψ)
Σ
K_4
$\frac{1}{\Delta t}$
Interval Between GPS Fixes
Note: Difference taken before integrator reset.

Figure 2: Twelve-State Velocity-aided Navigation Filter [BACH96]

Since the linear accelerometers are being used to measure the local vertical, if they are rotated around an axis parallel to the local vertical vector, their outputs will not change. Thus, an undetectable drift of the linear accelerometer / angular rate sensor package may occur about the local vertical except at the Earth's magnetic poles. This leads to a buildup of error in the azimuth estimation which cannot be overcome by the addition of the angular rate sensors, as they have their own bias errors which cause drift.

To counteract this, a flux-gate compass is used in combination with the linear accelerometers and angular rate sensors. Since the compass is sensitive to the Earth's magnetic field and not the gravity vector (local vertical), it can detect rotations about the local vertical, and can compensate for the azimuth drift not detected by the linear accelerometers. Thus, if a body is either stationary or moving at a constant velocity (zero acceleration) then the combination of linear acceleration and flux-gate compass sensors will provide accurate indication of the body's orientation with no singularities.

However, due to linear accelerometer slosh, the above combination of linear accelerometers and flux-gate compass is a low-frequency (long term) orientation solution only. If the attached body is experiencing accelerations other than gravity, the indicated orientation will be incorrect, depending upon the magnitude of the sensed accelerations. To compensate for this, angular rate sensors provide the high-frequency component of the indicated orientation. When the outputs of the entire sensor package are combined using the filter network shown in Figure 2, a stable, accurate estimation of the AUV's Euler angles (orientation) results. [MCGH95 and BACH96]

D. SUMMARY

In summary, a stable orientation vector may be obtained for any body in an Earth-fixed reference frame by the utilization of an orthogonal set of linear accelerometers, angular rate sensors and flux-gate magnetometers. The linear accelerometer outputs are averaged of the long term to yield the gravity vector. The short term components of the angular rate sensor outputs are combined with linear accelerometer outputs by the use of a complementary filter to compensate for linear accelerometer "slosh". And, finally, the flux-gate magnetometer outputs are utilized to compensate for the azimuth drift of the linear accelerometer / angular rate sensor combination.

The next chapter discusses the use of the above combination of inertial and magnetic sensors in human body motion capture. Chapter V then specifically addresses the use of a three-axis fluxgate magnetometer for azimuth drift compensation.

IV. APPLICATION OF INERTIAL SENSORS TO HUMAN BODY MOTION CAPTURE

A. HUMAN BODY TRACKING PROBLEM RE-STATED

Having established that inertial sensors can be used in a non-standard way to give a stable angular orientation estimate of an attached body, the following question remains: How can these techniques be applied to human body motion capture as an interface paradigm for synthetic environments?

First, a complete definition of the problem is necessary. What is desired is a system capable of tracking a human body as a collection of 15 rigid segments in real-time while attempting to meet the following goals:

1) Be unobtrusive (i.e. not encumbering to the user).
2) Allow virtually unlimited range of use / workspace size.
3) Be insensitive to electromagnetic, acoustic, and other forms of interference.
4) Be untethered.
5) Be capable of tracking in any environment.
6) Be accurate, linear and stable with no singularities.
7) Be reasonably cost-efficient in the long term (i.e. after initial development costs).

Obviously, this is a very ambitious set of goals. As stated in [FREY95], there is no system currently on the market which is capable of meeting all of these goals. The most widely used systems available for real-time human body tracking are the electromagnetic systems produced by Polhemus, Incorporated, and Ascension Technology Corporation.[FREY95] However, these systems do not allow unlimited range tracking or workspace size. They are tethered, incapable of tracking in all environments, and are sensitive to electromagnetic interference. In addition, all of the systems produced by either company are rather expensive, at least at the present time.

The author believes that a method utilizing inertial sensors, as described in Chapter III, is readily applicable to real-time human body tracking and will revolutionize the way in which human body tracking is done. The inertial system described below does not suffer from any of the current drawbacks of human body tracking systems as described in [FREY95] except, perhaps, being encumbering to the user. This, of course, depends upon the specific implementation of the inertial system.

Understanding the problem, the information desired from the system must next be defined. The author has proven, through human body modeling simulations written in LISP and C++ (discussed in Chapter VI), that an entire human body can be reasonably modeled as an articulated collection of 15 rigid body parts (head, torso, hips, upper legs, lower legs, feet, upper arms, lower arms and hands), and that nothing more than 15 sets of Euler angles (Earth-fixed azimuth, elevation and roll) is required to completely pose the entire body. This assumes a general synthetic environment application. Specific applications may require modeling of additional body parts such as separate shoulders, detailed hands or toes. If these additional body parts must be modeled, they may not be trackable by *any* generic motion capture system. Thus, another method of determining their orientations may be necessary (for instance, the use of a system to track detailed hand and finger motions).

The output which must be generated by the inertial tracking system is, therefore, a collection of 15 sets of Euler angles (azimuth, elevation and roll); one set for each of the 15 body parts being tracked. To provide this data, each body part's sensor package must be capable of sensing 3 axes of linear acceleration, 3 axes of angular rate and 3 axes of flux-gate magnetometer (described in Chapter V). The function of each of these sensors is described next.

The linear accelerometers, discussed in Chapter III, provide the low-frequency component of the Euler angle estimations, using the Earth's gravitational acceleration as a reference. However, linear accelerometers (being used to determine the direction of the local vertical) cannot detect azimuth rotations of a body about the axis of the Earth's gravitational field, typically considered to be the local vertical. For this reason, a three-axis flux gate magnetometer is used to compensate for rotations of the body about the local vertical. It can accomplish this because it measures orientation relative to the Earth's local magnetic field vector, which is not collinear with the local vertical. The combination of these two sensors yields a stable, low frequency estimation of the sensor package orientation. The angular rate sensors, as described in Chapter III, provide the high-frequency component of the Euler angle estimations to compensate for the slosh of the linear accelerometers. The low frequency component provided by the linear accelerometers compensates for the drift of the angular rate sensors.

In actuality, the outputs from the linear accelerometers are combined with those from the angular rate sensors by a complementary filtering arrangement. [BROW92] The result is composed of the low frequency contribution from the linear accelerometers and the high frequency contribution from the angular rate sensors. This data is used to estimate the pitch and roll of the rigid body being tracked. The estimates of pitch and roll are then combined with the outputs of the flux-gate magnetometer to estimate the rigid body's azimuth, as described in Chapter V. Finally, the three desired Euler angles are made available as the orientation estimate of the rigid body being tracked.

B. TRACKING A HUMAN BODY WITH INERTIAL SENSORS

Standard inertial sensors have, in the past, been fairly bulky. In fact, a reasonably accurate sensor package with low drift rate, manufactured by Systron-Donner for standard inertial navigation applications, weighs several pounds and is the size of a pint milk carton. Clearly, this type of sensor cannot be used for human body part orientation tracking.

What has transpired to make an inertial human body tracking system possible is the development of micro-machined linear accelerometers and angular rate sensors. There are several companies now manufacturing micro-machined inertial sensors, linear accelerometers and angular rate sensors, which can be used to produce the inertial sensor package described above in a much smaller package, with minimal power requirements. [FOXL94]

Fifteen inertial sensor packages of this type would be sufficient to accurately track an entire human body. The outputs of each of these sensors would be transmitted to a belt-mounted electronics package which would perform the data processing necessary to convert the inertial sensor outputs into the desired body part orientation Euler angles. These 15 sets of Euler angles could then be combined into a data packet and sent via wireless, radio frequency communication means to the host computer system for further processing. Alternatively, raw sensor data could be transmitted with all computations being performed at the host computer interface. The author believes that it would be more efficient to process the raw sensor data into orientation vectors prior to transmission. This subject, however, is left for future work.

This system would serve as an ideal human interface to a synthetic environment simulation system. The user could move in natural ways while the system unobtrusively tracks his/her body's motions. The lack of a signal source or tether would allow virtually unlimited range of operation and the system would be virtually immune to the normal sources of signal interference. The system would be capable of tracking the human body in almost any environment a human can enter.

Unlike current image-based and optical motion capture systems, an inertial tracking system would not suffer from shadowing. Unlike current electromagnetic systems, an inertial system would not suffer from a limited tracking range and would not be susceptible to metallic device interference, with the possible exception of interference to the flux-gate magnetometers used to stabilize azimuth drift. And, unlike acoustic systems, an inertial system would not suffer from acoustic interference. Thus, it can be seen that, should this type of system be developed to its fullest extent, it would not suffer from any of the limitations of current motion capture systems, with the possible exception of user encumbrance. This limitation could be overcome as well by the development of application-specific electronics, making the system lighter, more compact and more energy-efficient.

In addition, the system would not be limited to tracking the human body. It could be used to track any object in which the orientation of the object with respect to an earth fixed coordinate system was desired.

C. AN INERTIAL TRACKING SYSTEM IN DETAIL

An inertial sensor package like the one described above, mounted on a rigid body, would produce nine outputs: Three outputs proportional to the rigid body's rate of rotation about its vertical, lateral and longitudinal axes; three outputs proportional to the rigid body's linear acceleration along its vertical, lateral and longitudinal axes; and three outputs proportional to the components of the Earth's magnetic field as sensed along the rigid body's vertical, lateral and longitudinal axes. These nine outputs provide all of the information required to synthesize a stable set of Euler angles describing the Earth-fixed orientation of the rigid body being tracked.

Following the work by McGhee et. al. [MCGH95 and BACH96], the first step in obtaining the Euler angles for a rigid body is to use the linear accelerometers' and angular rate sensors' outputs, combined using the complementary filtering arrangement shown in Figure 2, to synthesize the pitch and roll of the body. The inputs to the filter are actually the estimates of pitch and roll from the linear accelerometers and the Earth-fixed (Euler) pitch and roll rates from the angular rate sensors.

First, following a derivation by McGhee [MCGH96], the linear accelerometer data (x and y components of linear acceleration) is converted into instantaneous estimates of pitch and roll (with slosh errors) using the following formulas, respectively:

$$\theta_a = \arcsin \frac{a_x}{g} \qquad (4.1)$$

$$\phi_a = -\arcsin \frac{a_y}{g \cdot \cos\theta} \qquad (4.2)$$

Note that the roll estimate is not usable when pitch reaches 90° since, at that point, the denominator of Equation 4.2 becomes zero. this limits that approach of this thesis to orientations with pitch between $-\frac{\pi}{2}$ and $+\frac{\pi}{2}$. This is a potentially serious problem which can be solved by representing orientation using quatemions. [COOK92] Such an extension is beyond the scope of this thesis.

Next, since the angular rate sensors are attached to the body being tracked, they naturally provide indications of angular rate in body coordinates. This data must be converted into the Earth-fixed coordinate system (Euler rates) before it can be used in the complementary filter. To do this, a body-fixed-rate-to-Euler-rate transformation matrix is applied to the angular rate sensors' outputs, which generates the following data transformations:

$$\frac{d\theta}{dt} = \cos\phi \cdot q - \sin\phi \cdot r \tag{4.3}$$

$$\frac{d\psi}{dt} = \sec\theta \cdot \sin\phi \cdot q + \sec\theta \cdot \cos\phi \cdot r \tag{4.4}$$

$$\frac{d\phi}{dt} = p + \tan\theta \cdot \sin\phi \cdot q + \tan\theta \cdot \cos\phi \cdot r \tag{4.5}$$

where p, q and r are roll rate, pitch rate and yaw rate in body coordinates, respectively. These equations can be found in matrix form in [MCGH93].

The transformed roll rate, pitch rate and yaw rate data, with the linear accelerometer estimates of pitch and roll, are then fed into individual (one for pitch, one for roll) complementary filters as shown in Figure 3 below.

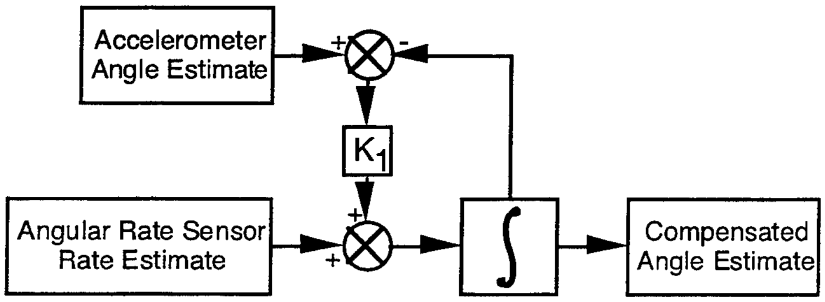

Accelerometer Angle Estimate
+
-
K1
+
Angular Rate Sensor Rate Estimate
+
∫
Compensated Angle Estimate

Figure 3: Individual Complementary Filter

The output of each filter is an instantaneous estimate of its respective Euler angle; pitch or roll. These estimates of pitch and roll are what is actually output from the system as the current pitch and roll of the rigid body. These estimates are also used in the conversion of the next set of angular rate data from body rates to Euler angle rates, the initial estimation of roll from the raw accelerometer data (Equation 4.2) and the estimation of azimuth when combined with the outputs from the flux-gate magnetometer for azimuth drift compensation (discussed in Chapter V).

The complementary filtering arrangement, shown in Figure 3, is described by the following S-domain (Laplace transform) equation:

$$\phi(s) = \frac{1}{s}\left(K_1 \cdot \phi_a(s) - K_1 \cdot \phi(s) + s\phi_s(s)\right) \tag{4.6}$$

When considering the responses of the above filter equation to the linear accelerometer inputs and the angular rate inputs individually, the following transfer functions result [MCGH95]:

$$\frac{\phi(s)}{\phi_a(s)} = \frac{1}{1+\mathrm{T}s} \quad \text{where} \quad \mathrm{T} = \frac{1}{K_1} \tag{4.7}$$

$$\frac{\phi(s)}{\phi_s(s)} = \frac{\mathrm{T}s}{1+\mathrm{T}s} \quad \text{where} \quad \mathrm{T} = \frac{1}{K_1} \tag{4.8}$$

Now, the superposition of the two responses yields the following result:

$$\frac{\phi(s)}{\phi_a(s)} + \frac{\phi(s)}{\phi_s(s)} = \frac{1}{1+\mathrm{T}s} + \frac{\mathrm{T}s}{1+\mathrm{T}s} = 1 \tag{4.9}$$

The importance of this result is that a perfect response is achieved with any value of K_1 for an ideal system. However; the realities of the real situation (noise, drift, 'slosh') imply that there is some optimal value for K_1. The constant K_1 determines the sensors' relative contributions to the final output. This value can only be determined through experimentation. This is beyond the scope of this thesis and is therefore a subject for future work.

In the above filtering discussion, only the filter equations for roll are shown. The filter equations for pitch and azimuth are identical. However, note that azimuth may not be reliably determined by this system due to bias errors in the angular rate sensors and the inability of linear accelerometers to detect rotations about the local vertical (gravity vector). Angular rate sensors, if integrated, can be used to estimate azimuth. However, with nothing to compensate for the drift caused by the bias errors of the angular rate sensors, a buildup of azimuth error results.

[MCGH95] uses a magnetic compass to obtain a low-frequency azimuth estimate for compensating the azimuth drift of the angular rate sensors. However, the magnetic compass used in this work does not function properly in all possible orientations. It is primarily designed to give azimuth only when oriented near the horizontal plane. If elevated or rolled too far, the compass ceases to function. For this reason, a three-axis fluxgate magnetometer has been chosen to perform azimuth drift compensation for the Artificial Vestibular System, as discussed in detail in Chapter V.

D. BIOLOGICAL ANALOG

The name that the author and McGhee have chosen to give the proposed system, "Artificial Vestibular System" results from the similarity of the system to the mammalian vestibular system. Every normal human head contains one complete vestibular system on each side, located in the vicinity of the inner-ear. The human vestibular system is composed, in part, by "semi-circular canals" and a "utricle". [HOWA66]

The semi-circular canals consist of three roughly orthogonal circular canals joined by a common cavity called the utricle. These canals are filled with endolymph fluid. Each canal contains an expanded passage, called the "ampulla", located near the point where the canal connects to the utricle. The ampulla contains the sensory epithelium, or "crista ampullaris".

The crista ampullaris is a protrusion of epithelium into the cavity of the ampulla. A multitude of sensory cilia project from the crista into a gelatinous mass called the "cupula". The cupula is formed to allow it to swing from side to side while it effectively blocks the flow of any fluid past the ampulla. The cupula thus forms a damped, self-centering pendulum.

When the semi-circular canals are rotationally accelerated, the movement of the canals relative to the endolymph fluid displaces the cupula and is sensed by the system as an angular acceleration. If the system is immediately decelerated (reverse rotational acceleration) the cupula will return toward its centered (non-accelerating) state and the system will sense this as a stopping of the initial turning motion. This is the normal state of most head motions; turning from a briefly stationary position to another briefly stationary position. If the entire movement does not last more than about 3 seconds, the human brain is able to accurately judge angular displacement.

If, however, the head is rotated at a constant angular velocity in one direction, the friction between the canals and the endolymph fluid will rapidly bring the endolymph fluid up to the rotational speed of the canals, allowing the cupula to return to its non-accelerating position. If the head is subsequently decelerated, the system will sense this as an angular acceleration in the opposite direction and the individual will experience a turning sensation and nausea. This is what happens when the human body becomes "dizzy".

The utricle is the common connecting chamber between the three semi-circular canals and is filled with the same endolymph fluid that is in the canals. The sensory body inside the utricle is called the macula. The macula is attached to the inside of the utricle on its anterior and medial walls. It consists of a mass of ciliated epithelial cells. The cilia extend outward into a gelatinous mass containing calcium carbonate particles called "otoliths". Linear accelerations cause displacements in the otoliths which are detected by the cilia.

Since the utricles are sensitive to the magnitude and direction of any linear acceleration, they are also sensitive to gravitational acceleration. For this reason, the vestibular system is able to sense its orientation relative to the local vertical (gravitational vector) when it is not being influenced by other linear accelerations.

The combination of the semi-circular canals and the utricle of the human vestibular system normally allows a human body to adequately sense its spatial orientation. The three semi-circular canals and utricle perform similar functions in the human vestibular system to that of the angular rate sensors and the linear accelerometers in the AVS. Thus the system proposed by the author and McGhee can be accurately referred to as an Artificial Vestibular System.

E. SUMMARY

To make synthetic environments more useful, some type of an unobtrusive, real-time human body tracking method is desirable. Due to the limitations of current motion capture technology, there is no good solution to this problem.

An inertial sensing system (Artificial Vestibular System) would not suffer from the drawbacks of the current motion capture technology. This is because the system has no signal source to be interfered with or to impose movement restrictions on the user. It uses the Earth's gravitational and magnetic fields as a stable orientation reference. The technologies which have made this possible are the new micro-machined linear accelerometers and angular rate sensors being produced by several companies in the United States.

The method of combining the sensor outputs using the complementary filtering scheme described above has already been proven to work successfully in autonomous underwater vehicle testing by McGhee, et. al. [MCGH95 and BACH96] This theory can be readily applied to the tracking of human body limb segments for the purposes of synthetic environment interface.

Still, the system described in this chapter suffers from azimuth drift, or drift around the local vertical. The next chapter discusses a method of compensating for this azimuth drift using a three-axis fluxgate magnetometer.

V. FLUX-GATE MAGNETOMETER FOR AZIMUTH DRIFT COMPENSATION

A. RATIONALE AND MAGNETOMETER BASICS

As was discussed in Chapter IV, the use of a combination of linear accelerometers and angular rate sensors can yield a stable set of Euler angles describing the orientation of the rigid body to which they are attached. A problem resulting from the use of these sensors is the linear accelerometers' insensitivity to rotations of the sensor package about an axis collinear with the local vertical. In other words, the linear accelerometers cannot compensate for the drift around the local vertical caused by the angular rate sensor bias errors.

To compensate for this, another sensor must be used which has a sensitive axis that is not collinear with the local vertical. The only other type of sensor which is commonly available and uses a natural signal source for reference is a compass or, rather, a device which is sensitive to the Earth's magnetic field. The device that the author has chosen to use is a three-axis flux-gate magnetometer.

A one-axis magnetometer is a device which senses the component of the Earth's magnetic field aligned with its sensitive axis. In other words, when the sensing axis of the coil is parallel to the magnetic field's lines of flux, the maximum output voltage will result. When the central axis of the coil is perpendicular to the lines of flux, no magnetic field will be sensed and the minimum output voltage will result. This behavior defines a vector dot product relation, so the magnitude and sign of the induced voltage are derived from a cosine function of the angle between the coil axis and the magnetic field's lines of flux, allowing for bias and noise voltages.

The orientation of a sensing coil relative to the Earth's magnetic field may be determined, in one dimension, by the comparison of the voltage induced in the coil by the Earth's magnetic field and the expected maximum and minimum voltages. If the maximum and minimum magnitudes of induced voltage are known, then the angle between the Earth's magnetic field and the coil axis may be estimated.

One magnetometer coil, however, will yield only a partial solution. It can be shown that two orthogonal magnetometer coils are sufficient to produce a complete solution for orientation provided the magnitudes of the expected bias and peak-to-peak induced voltage are known. [MCGH96B] However, in the absence of any magnitude information about the Earth's local magnetic field strength, three orthogonal magnetometer coils are required for a complete solution with automatic calibration of expected bias and peak-to-peak induced voltages.

B. REAL MAGNETOMETER BENCH TEST

In order to study the operation of a magnetometer in detail, the author bench tested a real magnetometer. This study was necessary to ensure that the author's simulation of a magnetometer would be correct, and that the author's development of azimuth estimation code would be valid for use with a real magnetometer.

The magnetometer available for the experiment was a model 9200C three-axis flux-gate magnetometer manufactured by Develco [DEVE86], serial number 1625-562. The power supply used to drive the magnetometer was an in-house bias box consisting of three nine-volt batteries and an on/off switch. A Micronta digital multi-meter was used for initial sampling of the magnetometer outputs. In the final data-taking efforts, a portable PC with installed multi-channel analog-to-digital converter interface hardware was used for automatic data logging.

The first step in the bench test was to build a bench test table which was free of metallic objects and had the capability to orient the magnetometer in three degrees of freedom (DOF). To simplify this process, the author chose to limit the orientation of the magnetometer to certain pitch and roll values, but to allow the magnetometer to be oriented at any azimuth. This allowed the author to use wooden blocks, cut at precise angles, to orient the magnetometer at the required pitch and roll values. The author chose to use pitch and roll values of 0°, ±30°, ±45°, and ±60°.

The table which was used contained a small amount of structural metal. It was found that the metal in the table adversely effected the measurement values if the magnetometer was moved relative to the location of the metal. Since this was the only table available, the author chose to continue to use it, but to constrain the movement of the magnetometer to be in the center of the table, away from the table's metallic parts. Eventually, it was decided that the magnetometer should be elevated above the table to minimize the effects of the structural metal. The data taking experiments were accomplished with the magnetometer elevated above the table by approximately one foot by placing a small cardboard box in between the table and the magnetometer.

The conduct of the experiment proceeded with the magnetometer being attached to each of the angled wooden blocks individually and rotated through 360 degrees in azimuth, logging the three magnetometer output voltages at azimuth intervals of 15 degrees, beginning with 0° magnetic (magnetic north). Plots of the magnetometer output voltages for pitches of 0°, ±30°, ±45°, and ±60° and rolls of +30°, +45° and +60° can be found in Appendix A. Plots of the same data from the author's C++ magnetometer simulation for corresponding cases are also included in Appendix A.

The comparison of the two sets of plots reveals that, although the magnitudes of the magnetometer output curves are slightly different, the general shape of the curves is the same. Thus, the author's magnetometer simulation appears to be correct for the cases shown. The outputs may be made identical by the correct choice of bias and peak-to-peak voltage magnitudes in the simulated case. The author chose not to take the time to do this when the follow-on simulation work did not require it.

C. MAGNETOMETER SIMULATION

The simulation of a flux-gate magnetometer turned out to be a rather simple matter. The experimental work with the real magnetometer reveals that the response of each magnetometer coil to various orientations relative to the Earth's magnetic field is a sinusoidal function of the angle between the coil axis and the Earth's magnetic field axis. This realization makes the simulation of the magnetometer output voltages a simple matter of establishing a reference peak-to-peak voltage vector (V_{ptp}) for each of the magnetometer coils, transforming the vector from Earth-fixed coordinates to body-fixed coordinates using a three-by-three rotation transformation matrix (tMatrix) representing the three relative orientation Euler angles (Roll, Elevation and Azimuth) and adding the reference bias voltage to each of the simulated magnetometer output channels.

The code for the implementation of the simulated magnetometer is included in AppendixB. The Supporting graphics and animation header files are included in AppendixD. The results of the magnetometer simulation are included in Appendix A along with the corresponding real magnetometer bench test results for comparison.

D. MAGNETOMETER AZIMUTH ESTIMATION

Once the experimental data was available, the next step was to develop the code which would estimate azimuth given the pitch and roll estimates and the three magnetometer outputs. The author chose to use C++ as the target language due to its popularity among software development circles, familiarity and the availability of the author's graphical simulation application programmers interface described in Chapter VI.

Appendix B contains the majority of the C++ code for magnetometer azimuth estimation. The code included was initially developed using MetroWerk's Codewarrior development environment for the Apple Macintosh series of computer systems. It was then ported to the Silicon Graphics family of workstations, using the OpenGL graphics library for the three-dimensional graphics and X-Windows for the windowing routines. A large portion of the code which generates the windows was written by Doctor Michael Zyda, Head of the NPSNET Research Group at the Naval Postgraduate School, and was adapted with his permission. An initial prototype of the simulation code was written by McGhee in Lisp and was available for comparison with the C++ simulation.

The portion of the code responsible for positioning and orienting the test figures is the author's Hercules Articulated Body Modeling package. This application programmers' interface (API) was designed to allow the building, animation and display of any articulated body. It is discussed in more detail in Chapter VI. The C++ header files for this system are included in Appendix D. Their filenames all begin with 'HG' to distinguish them from other header files.

The first step in magnetometer azimuth estimation is to initialize the simulated magnetic field that will be used as a reference. The initial field vector, called rawMagField, contains a magnetic field which is directed solely in the negative-x direction. This direction is taken to be due south for the majority of the Earth. To this initial vector, the magnetic field deviation and dip angle matrices are applied to give the reference magnetic field vector a realistic direction. Since the simulation was originally written in Monterey, California, the local magnetic field deviation and dip values are used. These are approximately 15° and 60°, respectively. However, the author has discovered that the deviation correction is unnecessary since magnetic north can be used for reference just as easily as true north. The deviation correction code has been left in place to allow for future applications which might require it, but the author has set the deviation value to zero for the purposes of this application.

Application of the dip and deviation matrices results in the final field vector, called earthMagField. This vector is then normalized to remove any dependencies on magnetic field magnitude. The resulting earthMagField reference vector is a unit vector which points in the direction of the dip-angle-corrected magnetic field for the Monterey, California, area.

Finally, the estimates of expected magnetometer bias, minimum and maximum channel voltages are calculated. These estimates are used to filter the incoming magnetometer data, allowing it to be compared properly to the earthMagField vector. Careful selection of the expected bias voltage for each channel is necessary to ensure an accurate azimuth estimate. Section E of this chapter covers this subject in detail.

During operation, the EstimateAzimuth function is the function which does all of the work of azimuth estimation. The data required to estimate the azimuth are the elevation and roll of the magnetometer and the three magnetometer outputs. As discussed, this data is available from elevation and roll estimates using inputs from the other sensors.

First, the expected bias voltage for each magnetometer channel is removed from the magnetometer outputs and the magnetometer output vector is normalized. The only difference between the normalized, expected earthMagField vector (N) and the normalized magnetometer output vector (B) is a three-by-three rotation transformation matrix (R) representing relative azimuth, elevation and roll rotations between the expected and measured magnetic field vectors. [MCGH96B] Thus:

$$N = R \cdot B \tag{5.1}$$

The matrix (R) is represented as follows [CRAI89]:

$$R = R(\psi) \cdot R(\theta) \cdot R(\phi) \tag{5.2}$$

Assuming that estimates of the magnetometer's elevation and roll are available from other sources (which they are as described in Chapter IV), the homogeneous transformation matrix representing these rotations is:

$$R_{\theta\phi} = R(\theta) \cdot R(\phi) \tag{5.3}$$

thus:

$$R = R(\psi) \cdot R_{\theta\phi} \tag{5.4}$$

Thus, as soon as the real magnetometer outputs are received, they are stripped of the expected bias voltages, normalized (B) and transformed into earth-fixed coordinates (M) by the application of the elevation and roll transformations as follows:

$$M = R_{\theta\phi} \cdot B \tag{5.5}$$

Now, the only difference remaining between the transformed magnetometer output vector (M) and the reference earth magnetic field vector (N) is the relative azimuth transformation:

$$N = R(\psi) \cdot M \tag{5.6}$$

Equation 5.6 can be re-written as:

$$\begin{pmatrix} N_1 \\ N_2 \\ N_3 \end{pmatrix} = \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} M_1 \\ M_2 \\ M_3 \end{pmatrix} \tag{5.7}$$

Simplification of Equation 5.7 gives the following solutions for N_1 and N_2:

$$N_1 = \cos\psi \cdot M_1 - \sin\psi \cdot M_2 \tag{5.8}$$

$$N_2 = \sin\psi \cdot M_1 + \cos\psi \cdot M_2 \tag{5.9}$$

Matrix solution of these equations for $\cos\psi$ and $\sin\psi$ yields:

$$\begin{pmatrix} \cos\psi \\ \sin\psi \end{pmatrix} = \frac{1}{M_1^2 + M_2^2} \begin{bmatrix} M_1 & M_2 \\ -M_2 & M_1 \end{bmatrix} \begin{pmatrix} N_1 \\ N_2 \end{pmatrix} \tag{5.10}$$

Finally, $\cos\psi$ and $\sin\psi$ are used to determine the azimuth estimation as follows:

$$\psi = \arctan(\sin\psi, \cos\psi) \tag{5.11}$$

However, note that Equation 5.10 fails whenever $M_1^2 + M_2^2 = 0$. Manipulating Equations 5.8 and 5.9 yields:

$$N_1^2 = M_1^2\cos^2\psi + M_2^2\sin^2\psi - 2M_1M_2\cos\psi\sin\psi \tag{5.12}$$
$$N_2^2 = M_1^2\sin^2\psi + M_2^2\cos^2\psi + 2M_1M_2\cos\psi\sin\psi \tag{5.13}$$

Now, adding Equations 5.12 and 5.13 yields:

$$N_1^2 + N_2^2 = M_1^2 + M_2^2 \tag{5.14}$$

Since N is the normalized Earth's magnetic field vector in north, east and down coordinates, note that $N_1 = N_2 = 0$ only at the Earth's magnetic poles. Thus, solutions exist for the above azimuth estimate (Equations 5.10 and 5.11) for all locations on the Earth's surface with the exception of the regions around the north and south magnetic poles. [MCGH96B]

The above azimuth estimation algorithm is implemented in C++. The C++ source code is included in Appendix B.

E. MAGNETOMETER CALIBRATION

It has already been stated in this chapter that selection of the expected bias voltages has a great impact on the accuracy of the estimated azimuth. In the simulation that the author has coded in C++ (Appendix B), the author has chosen expected bias and peak-to-peak voltage values based on empirical observation during the bench testing of a real magnetometer. This method of setting the bias voltages can yield a fairly accurate solution initially. The author's choice of bias voltages (which were identical for all three magnetometer channels) yielded azimuth estimates with an average absolute error of around 2.3 degrees. While this may be suitable for some applications, including the AVS, the author believes that some method of automatic calibration of the bias voltages would yield a better real-time solution. To this end, the author developed a method of automatic expected bias voltage calibration which was designed to reset the expected individual channel bias voltages, each time an azimuth estimate was requested, if a set of boundary criteria was violated.

The boundary criteria chosen are the running maximum and minimum output voltages from each channel of the magnetometer. During each estimation, the current magnetometer outputs are passed to the azimuth estimation routine. The estimation routine calls the MagSelfCal function to set the expected magnetometer bias voltages if necessary.

The MagSelfCal function, as shown in Appendix B, updates running minimum and maximum magnetometer voltages for each channel. It uses these voltages as boundary conditions to test the current magnetometer outputs voltages. If the output voltage for a channel is either above the current maximum or below the current minimum, the maximum or minimum voltage, respectively, is set to the current output, and the bias voltage for that channel is re-computed as the average of the maximum and minimum voltages.

In this manner, each channel bias voltage can be adjusted, on the fly, to account for differences in day-to-day operation. By adjusting the bias voltages in real-time, there are no anomalies which could alter the bias voltages from day-to-day which cannot be accounted for.

This method relies on two assumptions: First, the initial expected bias and peak-to-peak voltages must be selected in a manner which will allow the MagSelfCal function to operate properly. The initial bias estimates must be as close to the real biases as possible to allow the initial system operation to be accurate. Also, the initial peak-to-peak voltage estimates, which are used to set the initial maximum and minimum values, must be selected so that the initial maximum and minimum values are within the expected bounds of the actual magnetometer maximum and minimum outputs. If they are not, the actual minimum and maximum magnetometer outputs will be ignored by the MagSelfCal function and will not be used to adjust the bias voltage properly. To accomplish this, the author used Equations 5.15 and 5.16, with 2.1 as the divisor, to initialize the expected maximum and minimum voltage values:

$$V_{\max} = V_{bias} + \frac{V_{ptp}}{2.1} \tag{5.15}$$

$$V_{\min} = V_{bias} - \frac{V_{ptp}}{2.1} \tag{5.16}$$

The initial results of this method of self-calibration were not promising. When used with the same magnetometer output data files and simulation used to generate the tables in Appendix C, the azimuth estimates generated were almost always worse, normally by a factor of about two. The author believes that this was caused by an insufficient data sample which allowed only a partial calibration. The output data taken during bench testing does not necessarily contain the actual maximum or minimum magnetometer output voltages.

This points out the second assumption made by the author: The author assumes that a magnetometer mounted on a user's body will pass through all six maximum and minimum output positions of its three channels in the first few minutes of operation. Clearly, if the magnetometer does not do this, the channel biases will not be adjusted properly, unless they are correctly estimated from the start. This is clearly what happened during the author's attempts to use the MagSelfCal function, as described above. The data files contain insufficient data to set all expected channel bias voltages properly, and the resulting azimuth estimates suffered.

This realization has led the author to the conclusion that some method must be used to accurately estimate the initial expected bias and peak-to-peak voltages. Since this cannot be done automatically, some method of human-assisted calibration must be performed. McGhee believes that this is as simple as manually positioning the magnetometer, watching for the maximum and minimum voltages of each channel, and setting the initial bias and peak-to-peak voltages for each channel using the following equations:

$$V_{bias} = \frac{V_{\min} + V_{\max}}{2} \tag{5.17}$$

$$V_{ptp} = V_{\max} - V_{\min} \tag{5.18}$$

Setting the initial values in this manner should allow the system to initially operate properly and should allow the MagSelfCal function to properly adjust the expected bias voltages for calibration during day-to-day operation.

F. SUMMARY

Comparison of the plots presented in Appendix A indicates that the author's magnetometer simulation accurately mimics the operation of a three-axis fluxgate magnetometer. Thus, the author's magnetometer can be used in the simulated Artificial Vestibular System (AVS) to provide the simulated voltage outputs of a magnetometer attached to an object in a virtual environment. The results of the simulated AVS may then be used to provide proof-of-concept for the design of the physical AVS.

The study of the data in Appendix C shows that the author's azimuth estimation routine gives a reasonably accurate estimate. Further experimentation with additional magnetometer data indicating more reasonable initial expected bias and peak-to-peak voltages should give a better azimuth estimate. Additionally, further work on self-calibration is necessary before 15 of the AVS sensors packages can be used to track the human body. It is the author's opinion that manual calibration of all sensors before each use is not feasible. This is left for future work.

While it is possible to track the human body and produce Euler orientation vectors, alone, for each of the participant's body parts, a question remains: Does tracking three DOF for each body part provide enough information for the host computer to synthesize a human body model for insertion into a virtual environment? The next chapter discusses the author's development of the "orientation-only" articulated body model.

VI. VALIDATION OF 'ORIENTATION-ONLY' ARTICULATED BODY MODELING

A. RATIONALE BEHIND 'ORIENTATION-ONLY' MODEL

The use of a motion capture system to track the Human body as an interface paradigm results in a stream of data (normally orientation and/or position data) entering the host computer system, indicating to the system the actions of the user. This data is used by the computer system to manipulate the virtual world in a way which corresponds to the actions of the user. If this is done properly, it can significantly contribute to suspending the disbelief of the user, or make them feel more like they are interacting with a real environment.

Part of the challenge of suspending the disbelief of the user lies in how the user visually perceives objects in the virtual environment, especially him or herself and other animated participants. Typically, the user and other participants are represented in the virtual world by what is currently being called an "avatar". This avatar becomes the alter-ego of the user for the duration of the user's stay in the virtual world and is the visual representation of the user which is presented to other participants.

The avatar may take any visual form that the system will graphically support. It is almost always composed of multiple rigid body segments, joined together in some fashion, which are allowed to be individually positioned and oriented for animation purposes. As long as the segments remain joined together and are animated in ways that are expected by other participants, the avatar will pass as a believable entity in the virtual environment. Bodies composed of joined rigid body segments are typically called "articulated" bodies.

Several paradigms exist for articulated body modeling in computer graphics. [BADL93] The differing methods of positioning and orienting the body parts is what typically separates these paradigms. Two differing schools of thought will be presented: Orientation-position method using inverse kinematics and orientation-only method.

1. Orientation-Position (Inverse Kinematics) method

One method of articulating rigid body segments involves the use of both orientation and position information to animate the rigid body segments of an articulated body. This method is typically utilized when the designer wishes to minimize the number of tracking sensors mounted on the user's body, or desires to minimize the amount of physical motion capture hardware to be used. This can have the effect of minimizing the cost of the motion capture system.

However, this cost advantage does not come without a price. The price paid is an increase in the complexity of the articulated body modeling software used. This software must now fill in the gaps in the motion capture data stream, generating orientation and position data for the user's body parts which are not tracked. The algorithms used to generate this information are collectively called inverse kinematics.

A typical configuration for minimizing the number of sensors used, when tracking the human body, is shown in Figure 4. [BADL93] Badler's system involves the use of four Polhemus electromagnetic receivers tracking in both position and orientation. This data is acquired at a rate of approximately 120 samples per second and is transmitted to the host computer system for articulated body animation.

Figure 4 shows that only the user's hands, head and upper body are tracked, leaving the orientation of the upper and lower arms to be determined using inverse kinematics. [BADL93 and CRAI89] Simply described, inverse kinematics involves the mathematical estimation of the orientation of un-tracked rigid body segments adjoining two tracked segments.

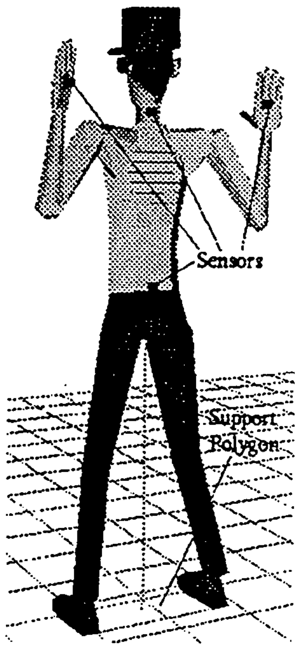


Figure 4: Minimal Tracking Sensor Configuration [BADL93]

In the case of Figure 4, assuming the motion capture system is capable of determining both position and orientation, the following are known quantities:

- position and orientation of the upper torso
- position and orientation of the head
- position and orientation of both hands

Thus, by assuming that these four tracked segments are absolutely rigid, the following information is known by reference:

- position of the shoulders
- position of both wrists

From this information, inverse kinematics is used to determine the following by numerical estimation techniques:

- orientation of the upper arms
- orientation of the lower arms

The solution to this problem, however, is not unique because there is not enough information to completely specify the position of the elbow. It can lie anywhere on a circle whose perpendicular axis is the line between the shoulder and wrist positions. To determine the position of the elbow, empirical research on human kinematics is used to predict the most likely position of the elbow, given the available information. This method almost always gives a reasonable solution, but it may not give the correct solution. The application will dictate whether this is acceptable or not.

One problem with using inverse kinematics is its computational complexity when compared with a system that tracks all of the user's body parts. Without position and orientation tracking information available for all body parts, the missing data must be determined using numerical methods and prediction. This can take a significantly longer time than tracking all of the body parts directly. Thus, it is the author's opinion that this method would result in a lower animation frame rate. Verification of this conjecture is left for future work.

Another problem with using both position and orientation information to synthesize an articulated body is the inherent inaccuracy of position tracking compared to orientation tracking. [SKOP96] During his work with Polhemus' sensors, Skopowski discovered that the position data from the system was of significantly lower quality than the orientation data. He hypothesized that this was due to the method in which these sensors accomplished position determination. Regardless of the reason, use of this data in the inverse kinematics determination of rigid body segment orientation results in inherent errors.

Along the same lines, image-based systems can be used to obtain reliable position data of multiple rigid body segments, but cannot be used for reliable determination of their orientation.

[FREY95] Thus, inverse kinematics will produce errors using data from this motion capture system as well.

The author has discovered that there is no motion capture system currently available which can reliably determine the position and orientation of enough independent objects to be used for tracking a 15 rigid segment articulated body (i.e. a human). [FREY95] Thus, either some reliable means of determining both position and orientation must be developed, or another articulated body modeling scheme must be used. In the author's opinion, the current methods of motion capture are far from being usable for fully tracking a human body. For this reason, the author has developed the Orientation-only articulated body model.

2. Orientation-Only Method

When all of the user's body parts are tracked individually, the orientation-only method may be used. This method does not involve the use of inverse kinematics. Thus, it is less computationally complex and, it has been the author's experience that a higher animation frame rate results. In addition, since only orientation data is required to fully synthesize an articulated body model, the demands on the motion capture system are not as extreme.

When only orientation data is required, electromagnetic motion capture systems are capable of providing real-time, consistent data for the synthesis of an articulated human body in a virtual environment. However, it is the author's opinion that, due to their inherent limitations (limited range, tether, interference), electromagnetic systems are not sufficient for human body tracking for virtual environment interface applications. The Artificial Vestibular System (AVS) is being developed for this reason. It is the author's opinion that the AVS will far surpass the capabilities of electromagnetic tracking systems.

To support the orientation-only capabilities of the AVS, the author realized that a method of synthesizing articulated bodies using only orientation data would have to be developed. For this reason, the author designed and coded the Hercules Articulated Body Modeling System.

B. HERCULES ARTICULATED BODY MODELING SYSTEM

The Hercules Articulated Body Modeling System was originally intended to provide a means of testing the AVS, once the system was available, but has grown into an entity itself. The basic goal of the system is to provide an application programmer's interface (API) which would make it ease for the application programmer (AP) to synthesize and animate articulated body structures. In addition, to allow testing of the AVS, the API provides the capability of using Euler angle or joint angle data from a motion capture system to orient each of the body's parts (segments). The Hercules C++ headers are included in Appendix D with descriptions of their use in Appendix E.

The basic premise of the Hercules system is that all articulated bodies may be represented by a hierarchy of individual segments, beginning with a unique root segment. Each segment relies on its parent for its global spatial positioning (3 DOF) and is provided its orientation (3 DOF) by the AP. No parent segment knows anything about its child segments, but every child segment knows which segment is its parent.

An articulated body is constructed by defining all of its individual segments as **RigidBody** class objects. Each object is provided its physical appearance by defining a list of vertices in its local coordinate system and defining a list of polygons based on these vertices. For this purpose, the author has created both **Point** (vertex) and **Polygon** classes. The **Point** and **Polygon** objects added to the segment's vertex and polygon lists using the **AddVertex** and **AddPolygon** methods, respectively.

The object is then given a visual appearance by assigning to it a material definition consisting of the standard OpenGL material specification of its ambient, diffuse, specular and shininess characteristics. The AP also provides the drawing method for the segment, selecting from **wireFrame**, **flatShaded**, and **smoothShaded**.

Once the physical appearance of the segment has been defined, the segment must be attached to its parent segment so that the Hercules API will know how to obtain its global spatial positioning information. If the segment is not joined to a parent, the AP must always provide its global spatial positioning information manually using the **SetPosture** method with 6 degrees of freedom. Once the segment has been joined to its parent, the Hercules API will always obtain its global spatial positioning information from the parent segment. The AP joins the segment to its parent by calling the **SetAttachmentPoint** method with a pointer to the segment to become the parent, the vertex within the parent at which the child segment will be attached, the attachment method (**absolute** or **relative**) and the axes of interest (if **relative** attachment is specified).

A **RigidBody** object is always rotated about its local origin by the Hercules API. When it is attached to a specific vertex of another segment, the origin of the child is always placed at the global spatial position of the specified vertex of the parent. If a segment is the root segment, by definition, it is not attached to another segment and must, therefore, always be provided its global spatial position and orientation by the AP.

An attachment method of **absolute** will allow a child segment to be oriented using global Euler angles regardless of the orientation of the parent. An attachment method of **relative** will allow the child segment to be oriented using Euler angles relative to the coordinate system of the parent. Essentially, **relative** attachment emulates the use of joint angles for orientation. The child segment is oriented relative to the parent, rather than relative to the world coordinate frame.

When **relative** attachment is used, the AP must tell the Hercules system which axes are relevant. For a three DOF joint, **azimuth**, **elevation** and **roll** are passed as parameters to the **SetAttachmentPoint** method during joining. For a one DOF joint, only one of the three **HGRotAxisDesignator** parameters is passed.

Once the body has been entirely defined, the AP must provide the spatial position and orientation (6DOF in global world coordinates) of the root segment, and the orientation of each child segment (3DOF). This is done by calling the **SetPosture** method of each segment with the appropriate data.

Once oriented, the **Transform** method of each segment must be called in hierarchical order. This is to ensure that body is properly positioned and oriented according to its location in the body hierarchy. In other words the **Transform** method must be called for the root segment first, followed by each child of the root, followed by each grandchild, etc. In general, a parent must be transformed before its children.

Once transformed, the segments may be drawn one at a time by calling the **RenderObject** method of the defined **ViewPoint** object with a pointer to each **Rigidbody** to be drawn. The Hercules API will then render the appropriate view of each segment according to its position and orientation and the position and orientation of the **ViewPoint** object. The OpenGL API will render each segment with the appropriate appearance, lighting, shading and hidden surface elimination as specified by the AP. [OPEN94]

An example application of the Hercules API can be found in Appendix F. This application consists of a fifteen-segment articulated human body. The body parts are constructed in the file HumanBodyParts.cpp. This file, along with HumanBodyParts.h, constructs a class definition for each of the body parts and codes the UpdatePosture method with a simple perambulation algorithm. This motion algorithm uses various sinusoids to produce a realistic walking or running motion to the human figure. The files HumanBody.cpp and HumanBody.h then define a human body class which gives central access points for all of the human body parts' functionalities.

C. SUMMARY

The author has shown, through C++ articulated body modeling code, that an entire human body can be built and animated using only three Euler angles for each body part. This result eliminates the need for human body motion capture systems to track the position of each body part. Thus, motion capture systems need only track 3 DOF orientation for each body part. The host computer system can then produce a human body model, representing the human body being tracked, using only orientation data to position all of the body parts.

The Hercules Articulated Body Modeling API can be used to build and manipulate any articulated body. The body can be built from raw vertex and polygonal data or from pre-built primitives included in the Hercules system. It can then be animated using only 3 DOF orientation data (either Euler angles or joint angles) for each body part.

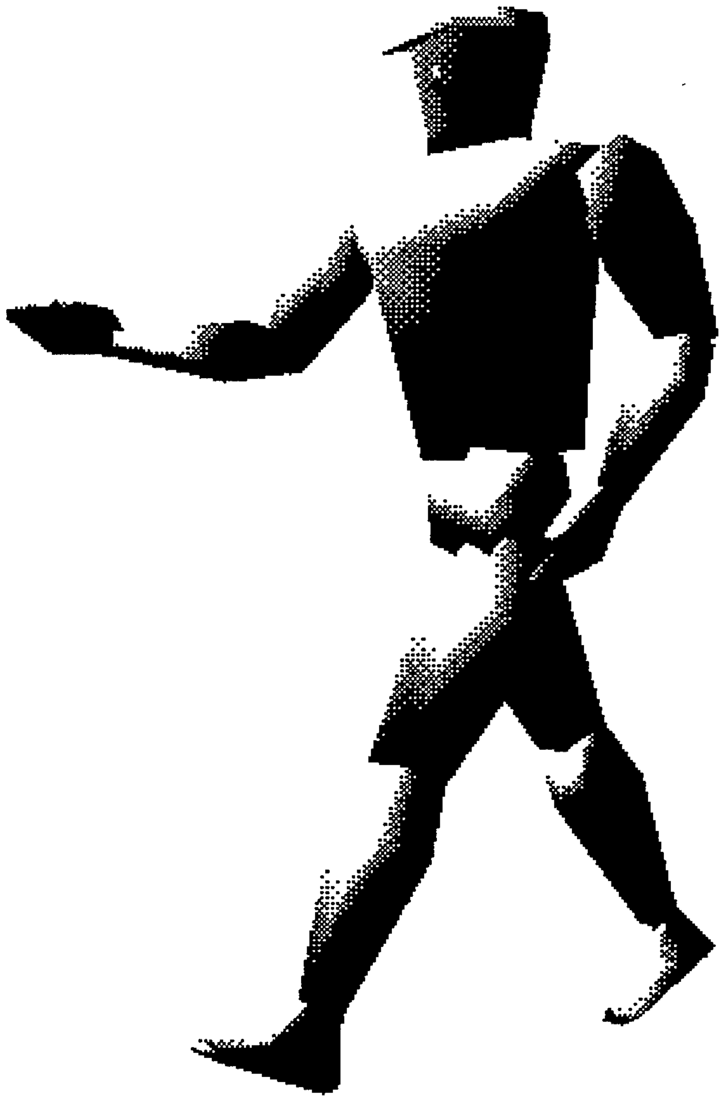


Figure 5: Hercules -- Hercules API Sample Application

VII. SUMMARY, CONCLUSIONS AND FUTURE WORK

A. SUMMARY AND CONCLUSIONS

All things considered, it is apparent that the world is moving into a time when the mouse and keyboard will no longer suffice as the predominant method for interfacing with synthetic environments and virtual worlds. Some method must be developed for interfacing human participants to the computer systems which generate these synthetic environments. The author believes that the new synthetic environment human interface paradigm will include some type of body tracking (motion capture) system for sensing the user's actions.

This thesis has discussed the various systems which are currently available to perform motion capture functions. Of these, the author has shown electromagnetic systems to be the only existing systems capable of reliable, accurate, real-time human body motion capture. [FREY95] However, these systems are encumbering to the user, spend too much time attempting to track objects in 6 DOF (position and orientation) when their position tracking leaves much to be desired [SKOP96], tether the user to the computer system, and are very susceptible to interference in the local vicinity.

The author has also shown in this thesis that articulated body modeling in a synthetic environment can be accomplished using only 3 DOF for each body part. This revelation allows motion capture systems to track only 3 DOF orientation for each body part and still provide the necessary human interface information. Thus, position tracking, which has typically been the least stable for electromagnetic trackers [SKOP96], is not necessary for synthetic environment interfaces.

Knowing that only orientation information is required allows motion capture sensors to be made smaller and lighter, thus reducing encumbrance of the user. These sensors can then be coupled with a system which requires no tether to the computer system, further reducing the user's movement restrictions.

While electromagnetic systems are good for motion capture, they suffer from significant drawbacks which render them inadequate for certain motion capture applications (i.e. long-distance motion capture and unrestricted freedom of movement). The primary reason behind this is that electromagnetic motion capture systems are sourced systems, meaning that they require an externally generated signal source to function. This source allows the electromagnetic systems to be subject to interference and range restrictions.

For this reason, the author has proposed a human body motion capture system (theAVS) based on inertial sensors, which requires no artificial signal source for motion capture. This would allow a user wearing the AVS unlimited freedom of movement and would allow the system to be virtually free of external interference. The user would no longer be tethered to the computer system and there would be virtually no range restrictions.

McGhee [MCGH95] and Foxlin [FOXL95] have shown that orientation-only motion capture using inertial sensors is both stable and reliable. Their methods of extracting orientation information from a combination of a linear accelerometers and angular rate sensors has proven to provide reliable orientation estimates. The only drawback of this method involves the angular rate sensor bias-generated rotational drift about the local vertical (azimuth drift). This discovery pointed out the need for incorporation of a third sensor which had a sensitive axis which was non-collinear with the local vertical.

To compensate for drift around the local vertical, the author and McGhee selected a three-axis flux-gate magnetometer. The magnetometer is sensitive to the Earth's magnetic field which is only collinear with the local vertical in the vicinity of the Earth's magnetic poles. Thus, for most regions of the Earth, a flux-gate magnetometer can be successfully used to compensate for azimuth drift.

This thesis has shown that a three-axis flux-gate magnetometer can be used to estimate the azimuth orientation of a body. This azimuth estimate can then be used to stabilize the entire inertial sensor package for rotational drift about the local vertical.

B. FUTURE WORK

While this thesis has shown that a stable and reliable 3 DOF orientation estimate can be obtained using a combination of inertial sensors, there is still much work to be done on the subject.

First, it is known that the system of orientation estimation used by McGhee [MCGH95] and described in Chapter IV fails when the elevation of the body being tracked reaches ±90°. This is due to the fact that the body-to-Euler angular rate transformation matrix (T-matrix in Figure 2) becomes singular there. The problem here is that an attempt is being made to determine the Euler angles of a tracked body. When the body reaches an elevation of ±90°, only the sum of azimuth and roll is defined; they are not individually determinable in this fashion.

To overcome this difficulty, a tracking method using quaternion filters is needed. Quaternions are not subject to singularities, so their use would allow unlimited tracking of a body with no limits on orientation. [COOK92] The development of the quaternion filter is left for future work.

The author has run various simulations of different aspects of the AVS and there has been much empirical research done by McGhee et. al. and Foxlin on inertial sensor orientation estimation in general. However, a sensor which will track the orientation of a body in 3 DOF at any orientation has not yet been constructed. The author is convinced that unlimited orientation tracking using inertial sensors, as described in this thesis, is possible and physically realizable in the next few years.

The logical course of action at this point is to begin specification of an inertial sensor package consisting of a three-axis linear accelerometer / angular rate sensor / flux-gate magnetometer combination. This sensor package should consist of micro-machined sensors and application specific integrated circuits to minimize the encumbrance to the user and to minimize the amount of data to be transferred out of the sensor package.

Once a sensor package has been synthesized, a system must be designed which is capable of processing the data from the inertial sensor packages in real-time (a minimum of 120 Hz) and providing it to the host computer system in way that is not encumbering to the user. In other words, the entire AVS, with the exception of the computer system interface hardware, must be body-mounted. This obviates the need for light-weight application specific electronics to do the orientation data processing on the user's body. The only data that should be sent from the body-mounted electronics is a stream of data packages, each containing the 3 DOF orientation estimate for all of the user's major body parts.

It is the author's opinion that an inertial tracking system like the AVS would uniquely satisfy the upcoming requirement of human body motion capture for the next synthetic environment interface paradigm without the drawbacks of the currently available motion capture systems.

APPENDIX A. PLOTS OF MAGNETOMETER OUTPUTS

The following plots are combinations of the three output channels of the flux-gate magnetometer. The first plot on each page is of data obtained during the bench testing of the real magnetometer. The second plot on each page is of data obtained by running the author's magnetometer simulator, described in Chapter V, with the same conditions as the real magnetometer data was obtained under (i.e. the same fixed elevation and roll).

Note that the curves are not labeled as to which magnetometer axis they represent. This is irrelevant, as the simulation can be made to produce any of the curves for any axis of the magnetometer. The important information is that the curves are similar in shape and phase differential for the given set of elevation and roll conditions.

Figure A.1 -- Real Magnetometer Data for Elevation = 0°, Roll = 0°

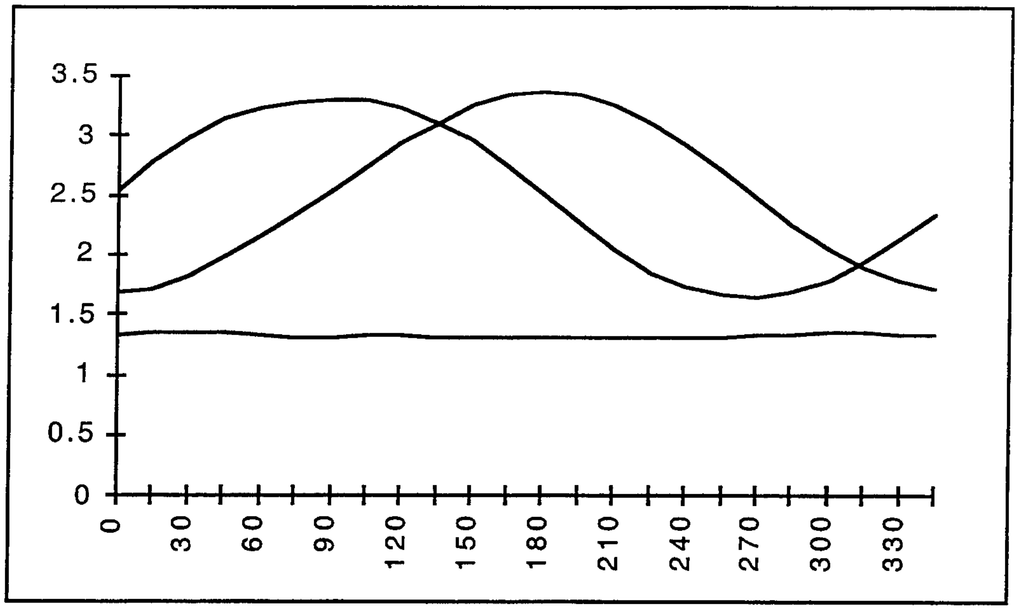


Figure A.2 -- Simulated Magnetometer Data for Elevation = 0°, Roll = 0°

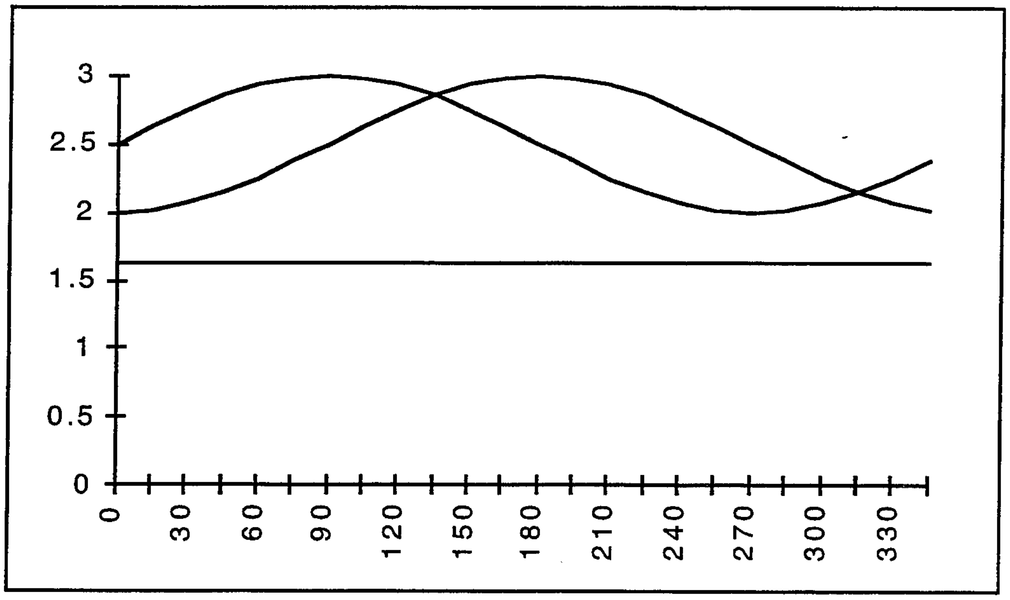


Figure A.3 -- Real Magnetometer Data for Elevation = +30°, Roll = 0°

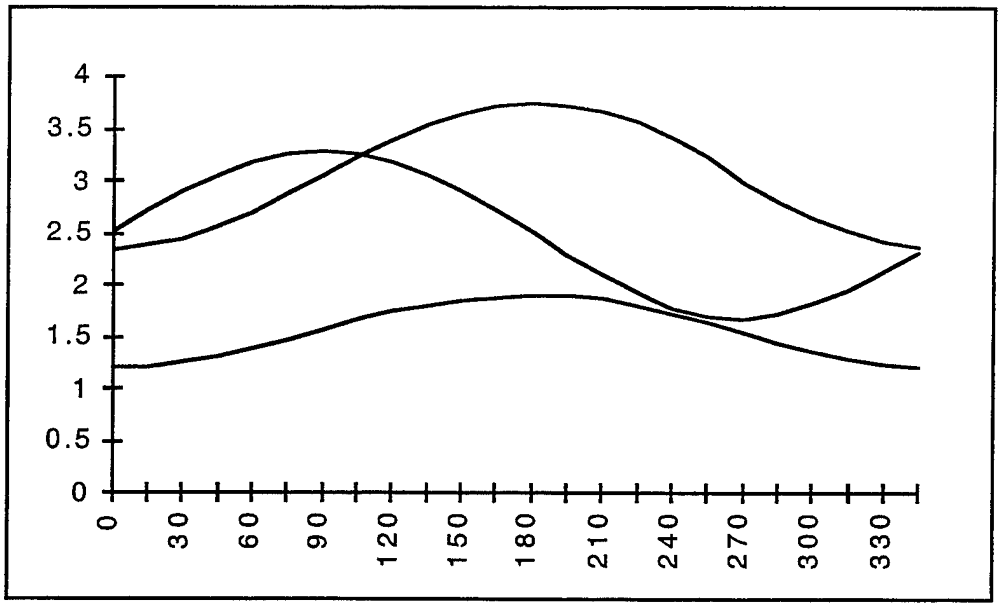


Figure A.4 -- Simulated Magnetometer Data for Elevation = +30°, Roll = 0°

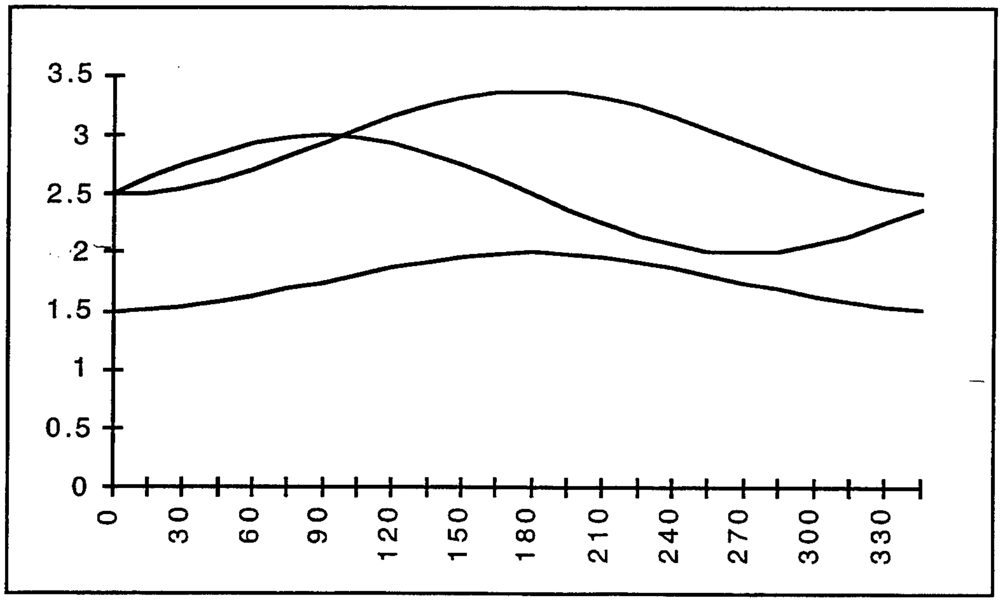


Figure A.5 -- Real Magnetometer Data for Elevation = +45°, Roll = 0°

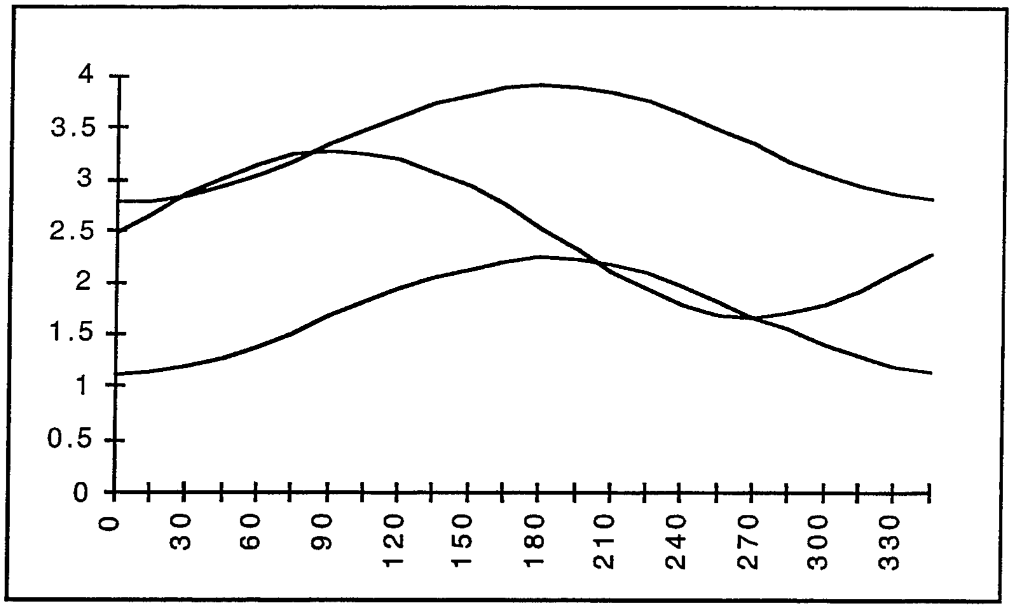


Figure A.6 -- Simulated Magnetometer Data for Elevation = +45°, Roll = 0°

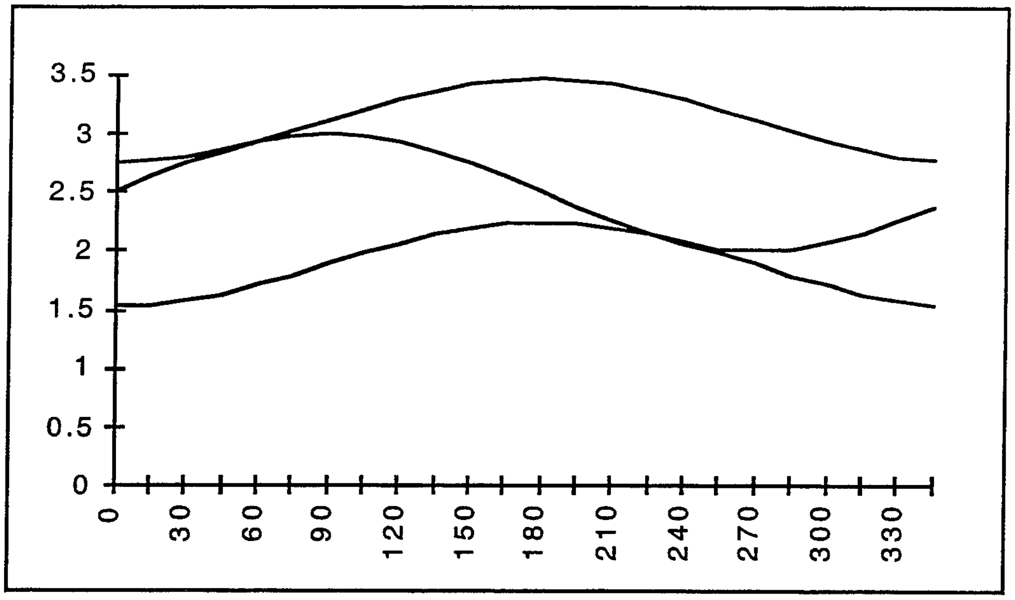


Figure A.7 -- Real Magnetometer Data for Elevation = +60°, Roll = 0°

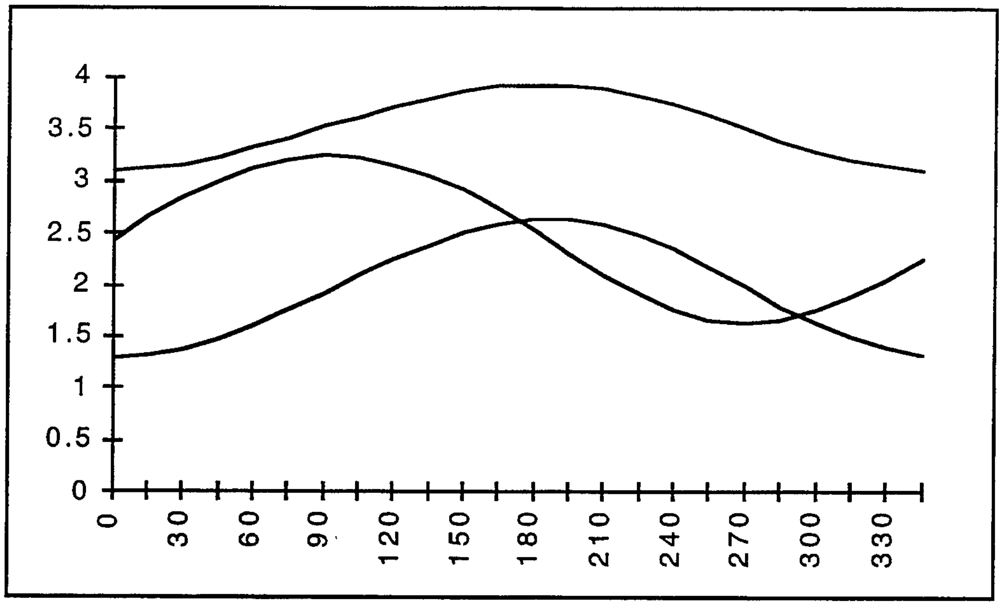


Figure A.8 -- Simulated Magnetometer Data for Elevation = +60°, Roll = 0°

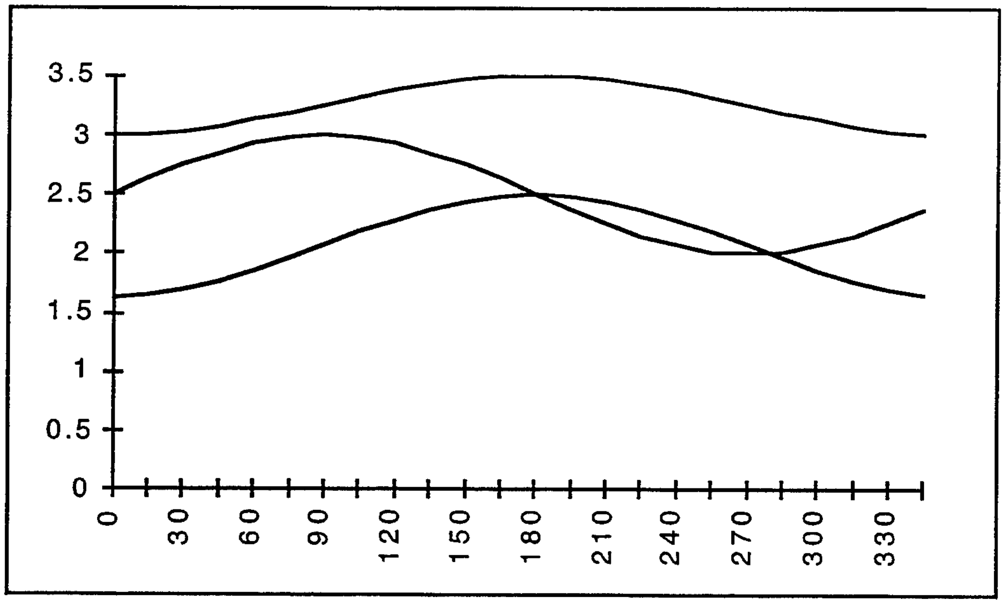


Figure A.9 -- Real Magnetometer Data for Elevation = 0°, Roll = 0°

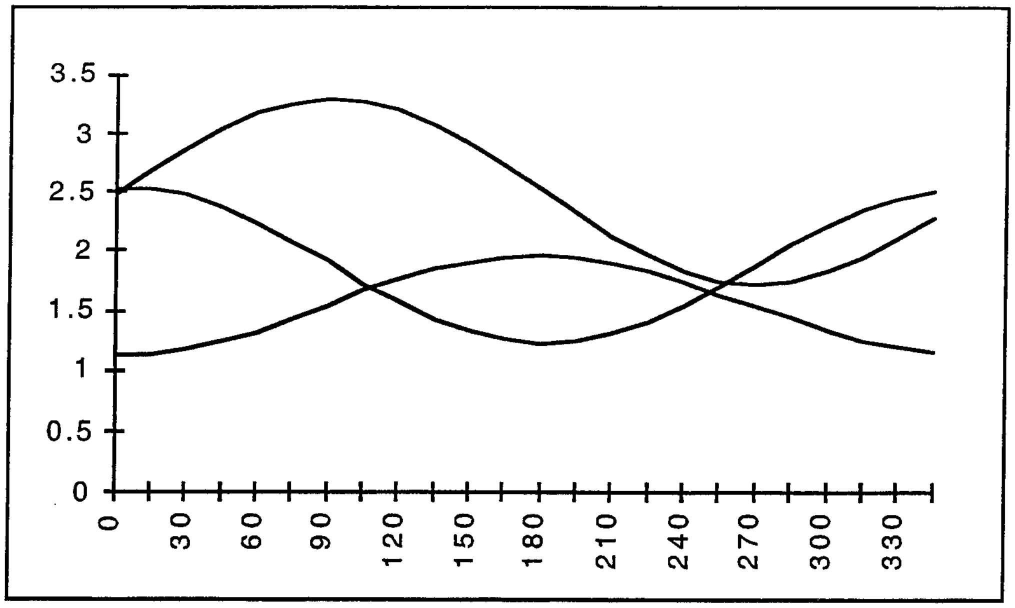


Figure A.10 -- Simulated Magnetometer Data for Elevation = -60°, Roll = 0°

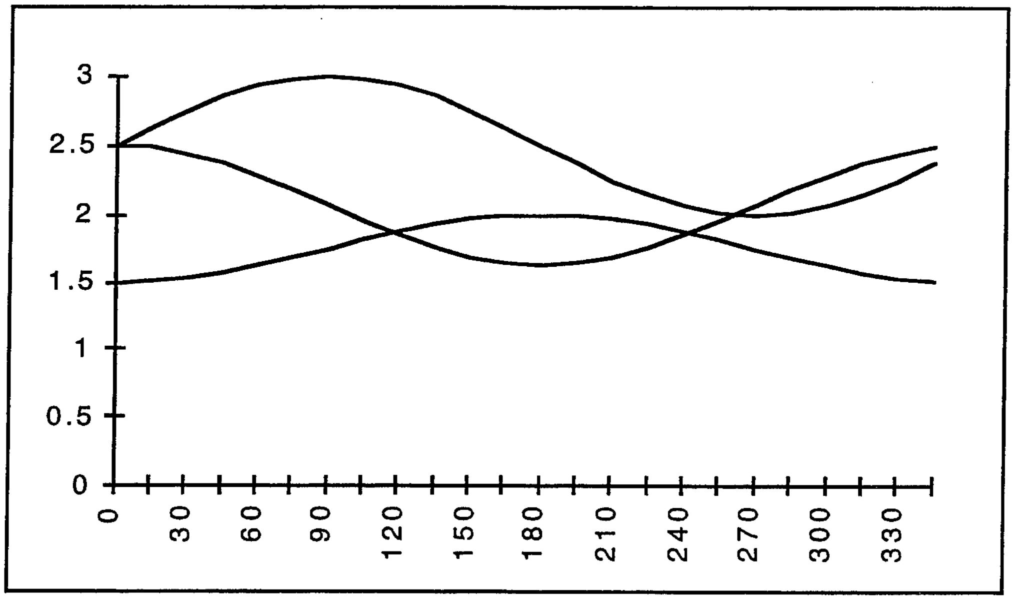


Figure A.11 -- Real Magnetometer Data for Elevation = -45°, Roll = 0°

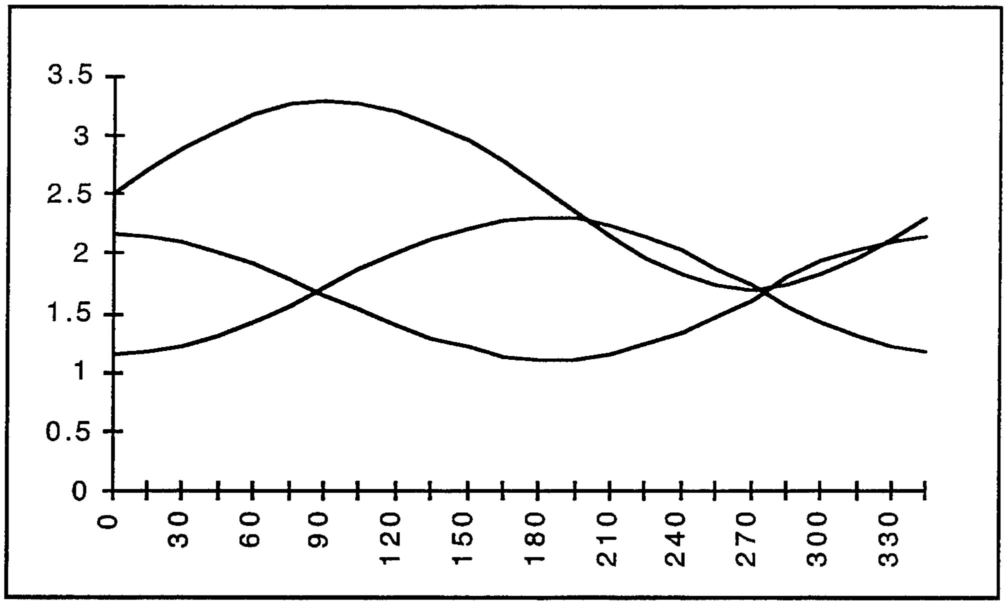


Figure A.12 -- Simulated Magnetometer Data for Elevation = -45°, Roll = 0°

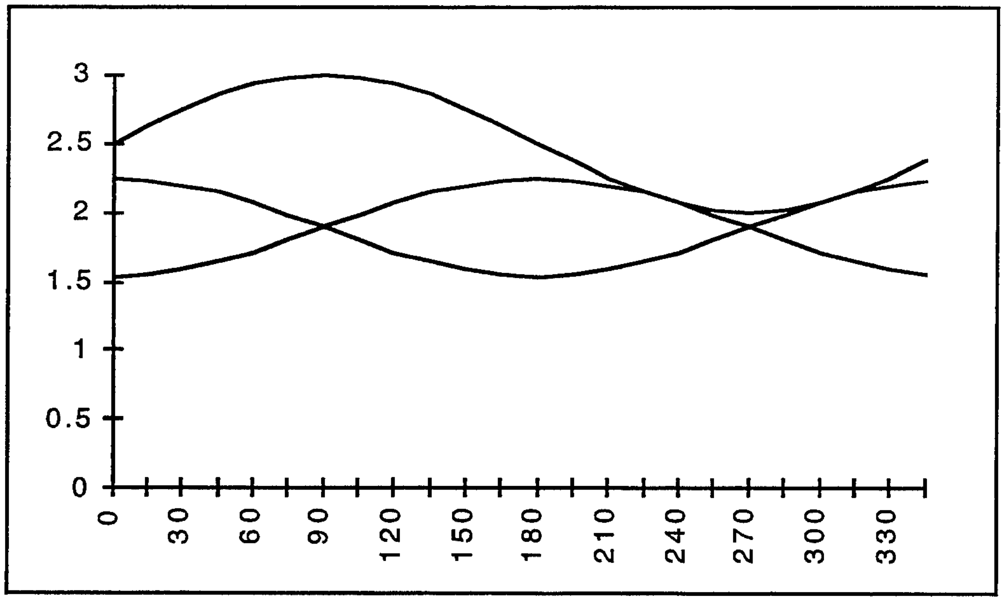


Figure A.13 -- Real Magnetometer Data for Elevation = -30°, Roll = 0°

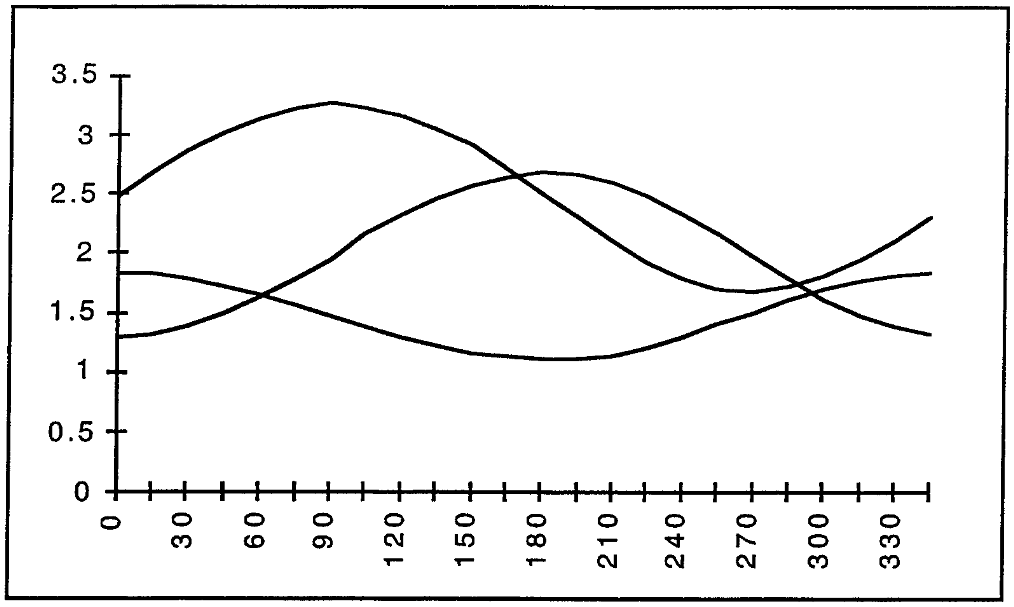


Figure A.14 -- Simulated Magnetometer Data for Elevation = -30°, Roll = 0°

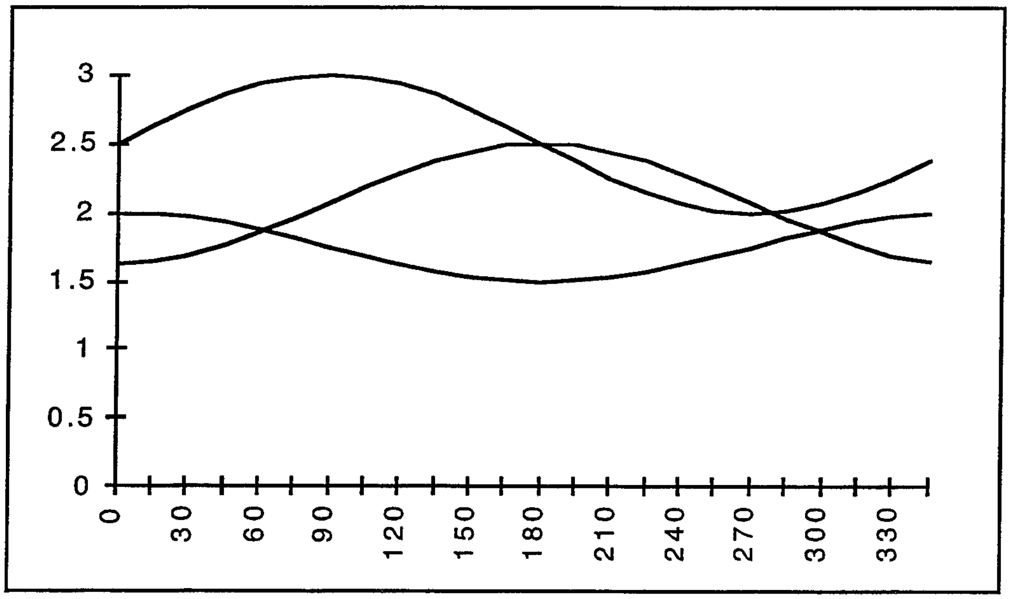


Figure A.15 -- Real Magnetometer Data for Elevation = 0°, Roll = +30°

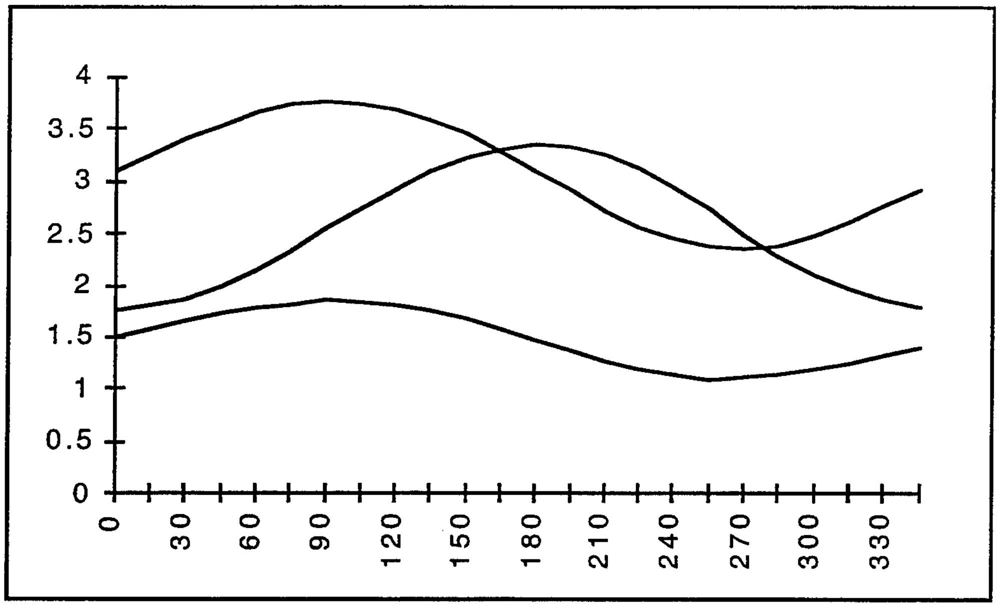


Figure A.16 -- Simulated Magnetometer Data for Elevation = 0°, Roll = +30°

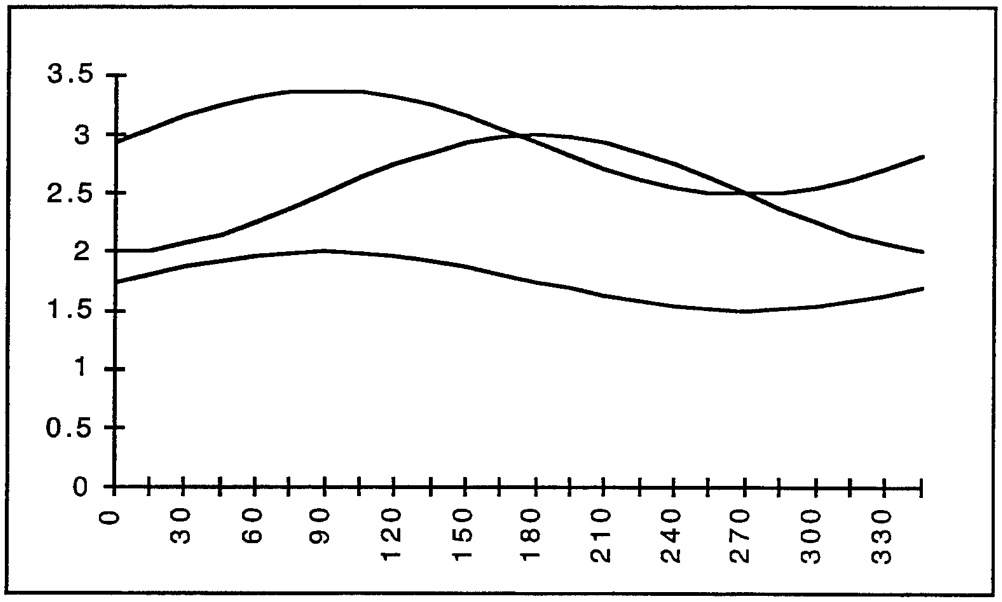


Figure A.17 -- Real Magnetometer Data for Elevation = 0°, Roll = +45°

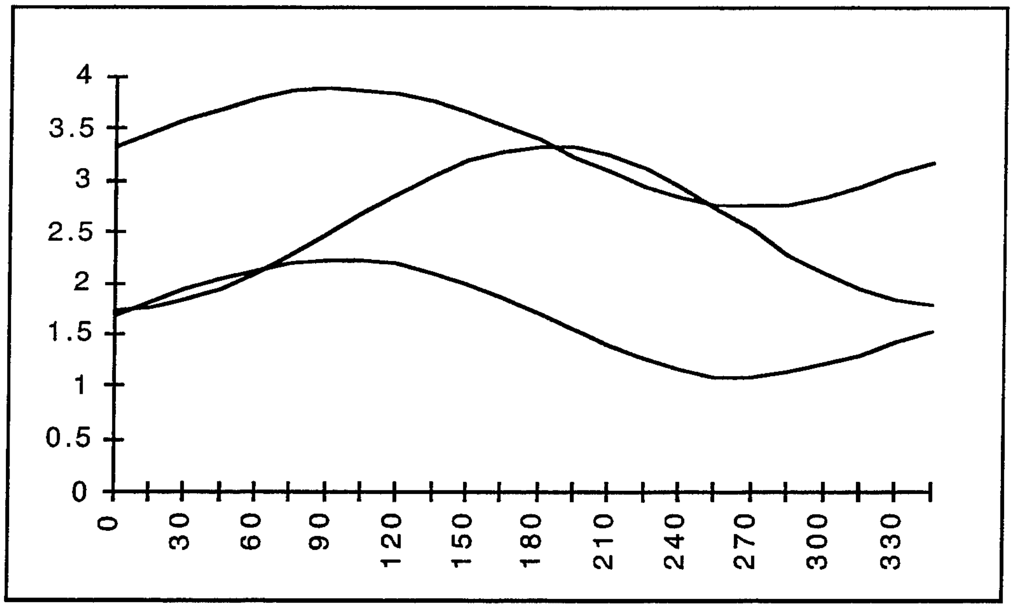


Figure A.18 -- Simulated Magnetometer Data for Elevation = 0°, Roll = +45°

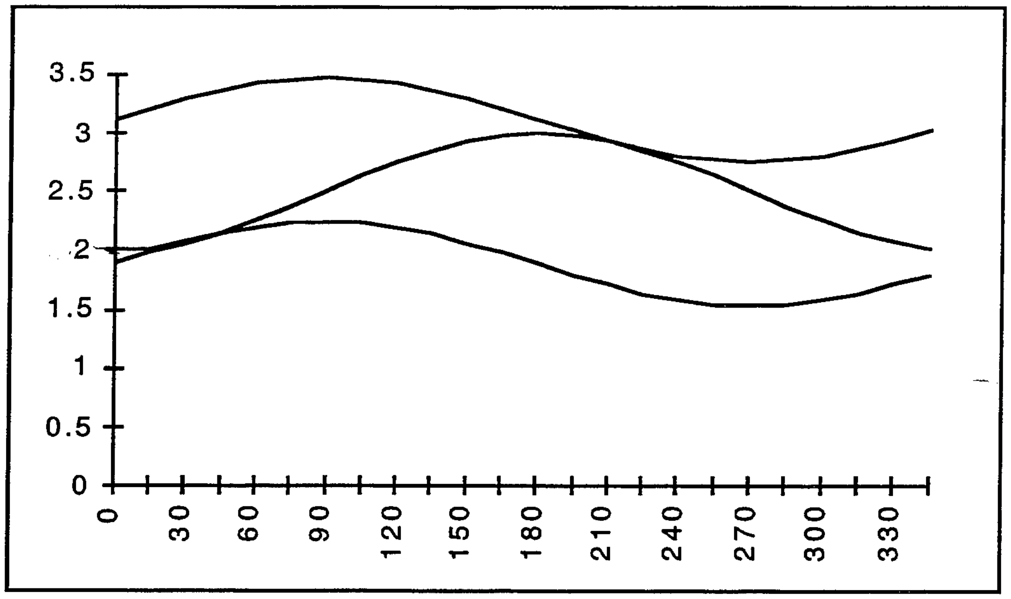


Figure A.19 -- Real Magnetometer Data for Elevation = 0°, Roll = +60°

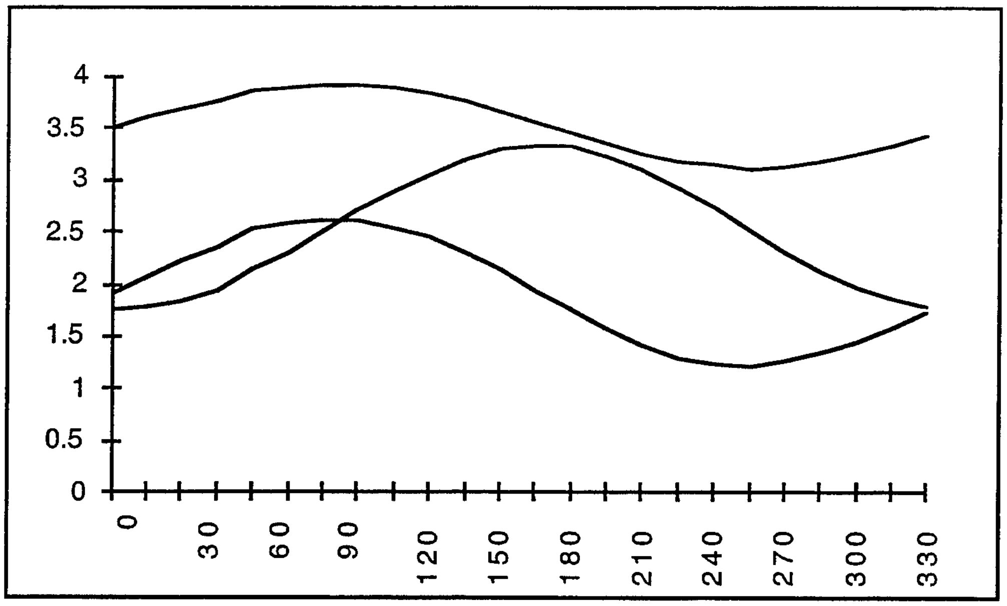
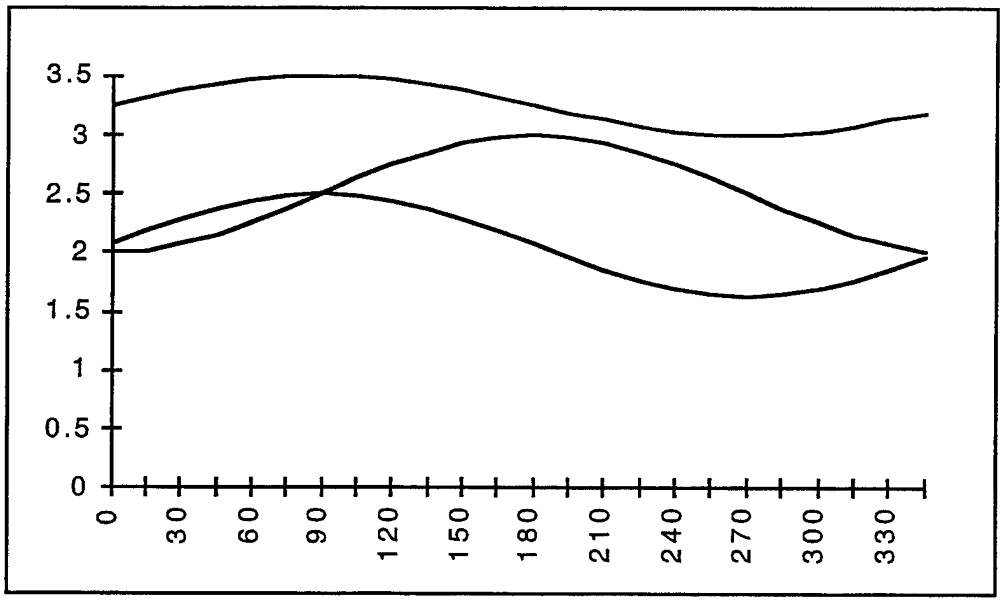


Figure A.20 -- Simulated Magnetometer Data for Elevation = 0°, Roll = +60°



APPENDIX B. MAGNETOMETER SIMULATION CODE

The following C++ code listings contain the majority of the source code necessary to produce a suitable magnetometer simulation and magnetometer azimuth estimates as described in Chapter V.

Section A contains the code for the magnetometer simulation. Note that the code for the magnetometer simulation is actually code for simulation of an entire inertial sensor package as described in Chapter IV, including a three-axis linear accelerometer, a three-axis angular rate sensor and the three-axis magnetometer.

Section B contains the code for magnetometer azimuth estimation. Note that the azimuth estimation routines are set up to use real magnetometer data from an input file rather than data from the simulated magnetometer of Section A. The author has run the azimuth estimation routines with simulated data as well. The estimated azimuth results were exactly equal to the true azimuth (error = 0) due to the consistency of the simulated data. Also note that most of the driver code (such as windowing routines) has been stripped out of the code for brevity. The remaining code is that which is directly responsible for initialization and azimuth estimation calculations.

The C++ header files for supporting graphics and animation routines may be found in Appendix D. The descriptions of these support routines may be found in Chapter VI. The supporting code and libraries, in their entirety, are available on request from the NPSNET Research Group at the Naval Postgraduate School.

A. Magnetometer Simulation Code

1. File: HGInertialSensor.h

```
/******************
   HGInertialSensor.h -- Simulated Inertial Sensor class
   Written by Wil Frey
******************/


#pragma once

#include <fp.h>
#include "HGtypes.h"
#include "HGMatrix.h"
#include "HGPoint.h"
#include "HGRigidBody.h"

// inertial constants
const float gravity = 9.81; // meters per sec squared
const float MagnetometerPToP = 1.6;
const float MagnetometerBias = 2.5;
const float bFieldDipAngle = -1.0472; // radians
const float bFieldDevAngle = 0; // radians

// sensor characteristics
const float LinearAccelBias = 0.0; // g
const float LinearAccelRMSnoise = 0.0; // g RMS
const float AngularRateInstability = 0.0; // degrees per hour
const float AngularRateRMSnoise = 0.0; // degrees RMS

// Inertial Sensor Package data structure
struct HGISensorDataType
{
   HGPoint AngularRate;
   HGPoint LinearAccel;
   HGPoint Magnetometer;
};

// Simulated Inertial Sensor Class
class HGInertialSensor
{
   public:

      HGInertialSensor(HGRigidBody *initBody, int initPointNum);
      HGISensorDataType *Poll(float time); // provide simulated sensor outputs
      HGPoint *GetEarthMagField() {return &magField;}
      HGRigidBody *GetBody() {return body;}
      HGISensorDataType *GetCurrentData() {return &currentState;}
      ~HGInertialSensor() {}
```

```
    private:

        HGRigidBody  *body;          // attached to which body
        int          bodyPointNum;   // attached where on body

        // Earth's mag field data
        HGPoint      magField;

        // last-state variables
        short        initState;      // initialization state of sensor (= 0,1,2)
        HGState6f    lastPosture;    // last _sensor_ position and orientation
        HGPoint      lastVelocity;   // last _sensor_ linear velocity
        float        lastTimeStamp;  // last timestamp

        HGISensorDataType currentState; // current sensor outputs
};
```

2. File: HGInertialSensor.cpp

```
/**********
HGInertialSensor.cpp -- Simulated Inertial Sensor class
Written by Wil Frey
***********/


#pragma once


#include "HGInertialSensor.h"


HGInertialSensor::HGInertialSensor(HGRigidBody *initBody, int initPointNum)
{
    HGMatrix  bFieldDevMatrix(4,4), bFieldDipMatrix(4,4);
    HGPoint   rawMagField;
    float     cdip, sdip, cdev, sdev;

    body = initBody;
    bodyPointNum = initPointNum;

    // set sensor state so that first two polls initialize all variables
    initState = 0;

    // set raw earth mag field vector
    rawMagField.x = 0;
    rawMagField.y = 0;
    rawMagField.z = -(MagnetometerPToP / 2);

    // calculate dip and deviation compensation matrix
    cdip = cos(bFieldDipAngle);
    sdip = sin(bFieldDipAngle);
    cdev = cos(bFieldDevAngle);
    sdev = sin(bFieldDevAngle);

    bFieldDevMatrix.SetElement(0,0,cdev);
    bFieldDevMatrix.SetElement(0,2,-sdev);
    bFieldDevMatrix.SetElement(2,0,sdev);
    bFieldDevMatrix.SetElement(2,2,cdev);

    bFieldDipMatrix.SetElement(1,1,cdip);
    bFieldDipMatrix.SetElement(1,2,sdip);
    bFieldDipMatrix.SetElement(2,1,-sdip);
    bFieldDipMatrix.SetElement(2,2,cdip);

    // calculate earth-fixed magField vector
    magField = rawMagField * (bFieldDevMatrix * bFieldDipMatrix);
}
```

```
// calculate linear accel, angular rate, magnetometer outputs
// returns pointer to current state if sensor initialized properly
// returns null pointer otherwise
HGISensorDataType *HGInertialSensor::Poll(float time)
{
    float deltaTime;
    HGMatrix bodyMatrix(3,3);
    HGMatrix tMatrix(4,4);
    HGState6f *posture = body->GetPosture();
    HGPoint *SensorPosit = body->GetTVertex(bodyPointNum);
    HGPoint linVelocity, linAccel, angVelocity;

    if (initState != 0) // at least one Poll has been done
    {
        // calculate delta time
        deltaTime = time - lastTimeStamp;

        // calculate earth-fixed linear velocity vector
        linVelocity.x = (SensorPosit->x - lastPosture.xp) / deltaTime;
        linVelocity.y = (SensorPosit->y - lastPosture.yp) / deltaTime;
        linVelocity.z = (SensorPosit->z - lastPosture.zp) / deltaTime;

        // calculate earth-fixed angular velocity vector
        angVelocity.x = (posture->el - lastPosture.el) / deltaTime;
        angVelocity.y = (posture->az - lastPosture.az) / deltaTime;
        angVelocity.z = (posture->rl - lastPosture.rl) / deltaTime;

        if (initState != 1) // at least two Polls have been done
        {
            // calculate earth-fixed linear acceleration vector
            linAccel.x = (linVelocity.x - lastVelocity.x) / deltaTime;
            linAccel.y = ((linVelocity.y - lastVelocity.y) / deltaTime) - gravity;
            linAccel.z = (linVelocity.z - lastVelocity.z) / deltaTime;

            // transform earth-fixed vectors into body-fixed vectors
            bodyMatrix = (body->GetHMatrix())->RotationMatrix4();
            tMatrix = bodyMatrix.Transpose();
            currentState.LinearAccel = linAccel * tMatrix;
            currentState.Magnetometer = magField * tMatrix;

            // add magnetometer biases
            currentState.Magnetometer.x += MagnetometerBias;
            currentState.Magnetometer.y += MagnetometerBias;
            currentState.Magnetometer.z += MagnetometerBias;

            // calculate body-fixed pitch angular velocity
            currentState.AngularRate.x =
                angVelocity.x * cos(posture->rl) +
                angVelocity.y * cos(posture->el) * sin(posture->rl);

            // calculate body-fixed azimuth angular velocity
            currentState.AngularRate.y =
                angVelocity.y * cos(posture->el) * cos(posture->rl) -
                angVelocity.x * sin(posture->rl);
            // calculate body-fixed roll angular velocity
            currentState.AngularRate.z =
                angVelocity.z - angVelocity.y * sin(posture->rl);

        }
    }

    // set 'last' variables with current values
    lastPosture.xp = SensorPosit->x;
    lastPosture.yp = SensorPosit->y;
    lastPosture.zp = SensorPosit->z;
    lastPosture.az = posture->az;
    lastPosture.el = posture->el;
    lastPosture.rl = posture->rl;

    if (initState != 0)
    {
        lastVelocity.x = linVelocity.x;
        lastVelocity.y = linVelocity.y;
        lastVelocity.z = linVelocity.z;
    }

    lastTimeStamp = time;

    if (initState < 2)
    {
        ++initState; // update initialization state variable
        return (HGISensorDataType *)0; // return NULL pointer if not stable
    }

    return &currentState;
}
```

B. Magnetometer Azimuth Estimation Code

1. File: SensorTest.cpp

```
/****************
    SensorTest.cpp -- Azimuth estimation test driver
    Written by Wil Frey
*****************/

// magnetometer biases
float MagnetometerBias = 2.5; // volts
HGPoint earthMagField, magBias, magMax, magMin;

// globals
float el=0, rl=0;
float estAzimuth;
float time, magX, magY, magZ;
int duration, count;


void InitMagnetometer()
{
    HGMatrix  bFieldDevMatrix(4,4), bFieldDipMatrix(4,4);
    HGPoint   rawMagField;
    float     cdip, sdip, cdev, sdev;
    float     magE;

    // set raw earth mag field vector
    rawMagField.x = 1;
    rawMagField.y = 0;
    rawMagField.z = 0;

    // calculate dip and deviation compensation matrix
    cdip = cos(bFieldDipAngle);
    sdip = sin(bFieldDipAngle);
    cdev = cos(bFieldDevAngle);
    sdev = sin(bFieldDevAngle);

    bFieldDevMatrix.SetElement(0,0,cdev);
    bFieldDevMatrix.SetElement(0,1,-sdev);
    bFieldDevMatrix.SetElement(1,0,sdev);
    bFieldDevMatrix.SetElement(1,1,cdev);

    bFieldDipMatrix.SetElement(0,0,cdip);
    bFieldDipMatrix.SetElement(0,2,sdip);
    bFieldDipMatrix.SetElement(2,0,-sdip);
    bFieldDipMatrix.SetElement(2,2,cdip);

    // calculate earth-fixed magField vector
    earthMagField = (bFieldDevMatrix * bFieldDipMatrix) * rawMagField;

    // normalize Earth-fixed magfield vector
    magE = sqrt((earthMagField.x * earthMagField.x) +
                (earthMagField.y * earthMagField.y) +
                (earthMagField.z * earthMagField.z));
    earthMagField.x /= magE;
    earthMagField.y /= magE;
    earthMagField.z /= magE;

    // set initial magnetometer calibration data
    magBias.x = MagnetometerBias;
    magBias.y = MagnetometerBias;
    magBias.z = MagnetometerBias;

    magMin.x = magBias.x - (MagnetometerPToP / 2.1);
    magMin.y = magBias.y - (MagnetometerPToP / 2.1);
    magMin.z = magBias.z - (MagnetometerPToP / 2.1);

    magMax.x = magBias.x + (MagnetometerPToP / 2.1);
    magMax.y = magBias.y + (MagnetometerPToP / 2.1);
    magMax.z = magBias.z + (MagnetometerPToP / 2.1);

    return;
}


float EstimateAzimuth(float inEl, float inRl,
                      float inMagX, float inMagY, float inMagZ)
{
    HGMatrix elR(4,4), rlR(4,4);
    HGPoint normMagField, transMagField;
    float magX, magY, magZ, magM;
    float sinPsi, cosPsi;

    float cel = cos(inEl);
    float sel = sin(inEl);
    float crl = cos(inRl);
    float srl = sin(inRl);

    // automatically adjust expected magnetometer biases
    // not being used due to current inadequacies
    // MagSelfCal(inMagX, inMagY, inMagZ);

    // build Rxy transformation matrix for known el and rl
    elR.SetElement(0,0,cel);
    elR.SetElement(0,2,sel);
    elR.SetElement(2,0,-sel);
    elR.SetElement(2,2,cel);

    rlR.SetElement(1,1,crl);
    rlR.SetElement(1,2,-srl);
    rlR.SetElement(2,1,srl);
    rlR.SetElement(2,2,crl);

    // acquire, remove bias from and normalize
    // simulated magnetometer data
    // Note that magX and magZ have been inverted to account
    // for actual magnetometer axes differences
    magX = -(inMagX - magBias.x);
    magY = inMagY - magBias.y;
    magZ = -(inMagZ - magBias.z);

    magM = sqrt((magX * magX) + (magY * magY) + (magZ * magZ));
    normMagField.x = magX / magM; // b1 in the derivation
    normMagField.y = magY / magM; // b2
    normMagField.z = magZ / magM; // b3

    // compute Earth-fixed magnetometer vector
    transMagField = (elR * rlR) * normMagField; // m in derivation

    // calculate estimated azimuth
    sinPsi = ((transMagField.y * earthMagField.x) -
              (transMagField.x * earthMagField.y)) /
             ((transMagField.x * transMagField.x) +
              (transMagField.y * transMagField.y));
    cosPsi = (earthMagField.x / transMagField.x) -
             ((transMagField.y / transMagField.x) * sinPsi);

    return (float) atan2(sinPsi, cosPsi);
}


// automatically calibrate expected magnetometer biases
void MagSelfCal(float inMagX, float inMagY, float inMagZ)
{
    short recal=0;

    // check for X bias recal
    if (inMagX > magMax.x)
    {
        magMax.x = inMagX;
      recal = 1;
    }

    if (inMagX < magMin.x)
    {
        magMin.x = inMagX;
      recal = 1;
    }

    if (recal)
    {
        magBias.x = (magMin.x + magMax.x) / 2;
      recal = 0;
    }

    // check for Y bias recal
    if (inMagY > magMax.y)
    {
        magMax.y = inMagY;
      recal = 1;
    }

    if (inMagY < magMin.y)
    {
        magMin.y = inMagY;
      recal = 1;
    }

    if (recal)
    {
        magBias.y = (magMin.y + magMax.y) / 2;
      recal = 0;
    }

    // check for Z bias recal
    if (inMagZ > magMax.z)
    {
        magMax.z = inMagZ;
      recal = 1;
    }

    if (inMagZ < magMin.z)
    {
        magMin.z =inMagZ;
      recal = 1;
    }

    if (recal)
        magBias.z = (magMin.z + magMax.z) / 2;

    return;
}
```

APPENDIX C. AZIMUTH ESTIMATION SIMULATION DATA

The following pages of data were obtained by running the azimuth estimation simulation code found in Appendix B with real magnetometer data obtained as described in Chapter VI. Note that the azimuth estimation routine actually produces azimuths between -180° and +180°. The estimated azimuth data in the following tables has been modified for ease of comparison. The data shown includes the true azimuth, the X-, Y- and Z-axis expected magnetometer bias voltages, the resulting azimuth estimate and the azimuth estimation error. The Average Error value is obtained by averaging the absolute value of all azimuth estimation errors.

Table C.1 -- Elevation = 0°, Roll = 0°

az	biasX	biasY	biasZ	estAz	error
0	2.527	2.527	2.527	358.634	-1.36554
15	2.527	2.527	2.527	12.5582	-2.44182
30	2.527	2.527	2.527	28.1156	-1.88437
45	2.527	2.527	2.527	42.2515	-2.74846
60	2.527	2.527	2.527	57.2764	-2.7236
75	2.527	2.527	2.527	72.2381	-2.76186
90	2.527	2.527	2.527	86.2813	-3.71867
105	2.527	2.527	2.527	101.066	-3.93399
120	2.527	2.527	2.527	115.865	-4.13542
135	2.527	2.527	2.527	130.187	-4.81306
150	2.527	2.527	2.527	145.583	-4.41736
165	2.527	2.527	2.527	162.454	-2.54573
180	2.527	2.527	2.527	178.85	-1.14983
195	2.527	2.527	2.527	194.664	-0.336151
210	2.527	2.527	2.527	210.834	0.83432
225	2.527	2.527	2.527	225.149	0.149078
240	2.527	2.527	2.527	240.133	0.133072
255	2.527	2.527	2.527	254.46	-0.539536
270	2.527	2.527	2.527	269.038	-0.96225
285	2.527	2.527	2.527	284.036	-0.963745
300	2.527	2.527	2.527	298.403	-1.59668
315	2.527	2.527	2.527	312.446	-2.55353
330	2.527	2.527	2.527	326.745	-3.25531
345	2.527	2.527	2.527	342.183	-2.81659

Average Error: 2.19917

Table C.2 -- Elevation = +30°, Roll = 0°

az	biasX	biasY	biasZ	estAz	error
0	2.527	2.527	2.527	0.692631	0.692631
15	2.527	2.527	2.527	14.7701	-0.229941
30	2.527	2.527	2.527	28.4169	-1.58309
45	2.527	2.527	2.527	42.2998	-2.70025
60	2.527	2.527	2.527	55.4996	-4.50045
75	2.527	2.527	2.527	69.3746	-5.62543
90	2.527	2.527	2.527	83.3228	-6.67721
105	2.527	2.527	2.527	100.68	-4.31989
120	2.527	2.527	2.527	115.462	-4.53841
135	2.527	2.527	2.527	131.073	-3.92725
150	2.527	2.527	2.527	146.596	-3.40439
165	2.527	2.527	2.527	161.963	-3.03661
180	2.527	2.527	2.527	178.463	-1.53712
195	2.527	2.527	2.527	194.696	-0.303909
210	2.527	2.527	2.527	210.923	0.922546
225	2.527	2.527	2.527	225.93	0.930344
240	2.527	2.527	2.527	241.676	1.67595
255	2.527	2.527	2.527	256.306	1.30585
270	2.527	2.527	2.527	271.202	1.20221
285	2.527	2.527	2.527	287.429	2.42917
300	2.527	2.527	2.527	303.07	3.06952
315	2.527	2.527	2.527	317.78	2.77994
330	2.527	2.527	2.527	332.637	2.63742
345	2.527	2.527	2.527	347.005	2.00488
Average Error: 2.58477					

Table C.3 -- Elevation = +45°, Roll = 0°

az	biasX	biasY	biasZ	estAz	error
0	2.527	2.527	2.527	4.05863	4.05863
15	2.527	2.527	2.527	18.4052	3.40518
30	2.527	2.527	2.527	32.3067	2.30669
45	2.527	2.527	2.527	46.5817	1.5817
60	2.527	2.527	2.527	60.2651	0.265087
75	2.527	2.527	2.527	74.3878	-0.612213
90	2.527	2.527	2.527	87.6729	-2.32713
105	2.527	2.527	2.527	102.179	-2.82083
120	2.527	2.527	2.527	117.64	-2.35997
135	2.527	2.527	2.527	133.063	-1.93739
150	2.527	2.527	2.527	148.277	-1.72319
165	2.527	2.527	2.527	164.374	-0.625839
180	2.527	2.527	2.527	180.813	0.813202
195	2.527	2.527	2.527	197.158	2.15839
210	2.527	2.527	2.527	212.956	2.95589
225	2.527	2.527	2.527	227.685	2.68477
240	2.527	2.527	2.527	241.668	1.66843
255	2.527	2.527	2.527	256.72	1.71982
270	2.527	2.527	2.527	271.56	1.55991
285	2.527	2.527	2.527	290.61	5.61023
300	2.527	2.527	2.527	305.948	5.94778
315	2.527	2.527	2.527	320.789	5.78925
330	2.527	2.527	2.527	335.347	5.34665
345	2.527	2.527	2.527	349.984	4.9837

Average Error: 2.71924

Table C.4 -- Elevation = +60°, Roll = 0°

az	biasX	biasY	biasZ	estAz	error
0	2.527	2.527	2.527	6.51264	6.51264
15	2.527	2.527	2.527	21.1648	6.16481
30	2.527	2.527	2.527	35.4032	5.40319
45	2.527	2.527	2.527	48.7785	3.77849
60	2.527	2.527	2.527	61.8341	1.83411
75	2.527	2.527	2.527	75.3703	0.370338
90	2.527	2.527	2.527	91.5399	1.53986
105	2.527	2.527	2.527	105.677	0.677109
120	2.527	2.527	2.527	119.839	-0.161194
135	2.527	2.527	2.527	133.783	-1.21677
150	2.527	2.527	2.527	149.611	-0.388748
165	2.527	2.527	2.527	163.754	-1.24602
180	2.527	2.527	2.527	179.423	-0.576614
195	2.527	2.527	2.527	195.262	0.261703
210	2.527	2.527	2.527	210.785	0.78479
225	2.527	2.527	2.527	225.705	0.705261
240	2.527	2.527	2.527	240.521	0.52066
255	2.527	2.527	2.527	256.413	1.41272
270	2.527	2.527	2.527	271.985	1.98514
285	2.527	2.527	2.527	288.598	3.59756
300	2.527	2.527	2.527	304.58	4.57983
315	2.527	2.527	2.527	320.462	5.46194
330	2.527	2.527	2.527	335.969	5.96912
345	2.527	2.527	2.527	351.037	6.03741

Average Error: 2.54942

Table C.5 -- Elevation = -30°, Roll = 0°

az	biasX	biasY	biasZ	estAz	error
0	2.527	2.527	2.527	2.802	2.802
15	2.527	2.527	2.527	18.6689	3.66891
30	2.527	2.527	2.527	33.8537	3.85372
45	2.527	2.527	2.527	48.374	3.37397
60	2.527	2.527	2.527	62.5658	2.56581
75	2.527	2.527	2.527	78.1094	3.10941
90	2.527	2.527	2.527	92.2729	2.2729
105	2.527	2.527	2.527	106.13	1.13042
120	2.527	2.527	2.527	121.136	1.13597
135	2.527	2.527	2.527	135.519	0.518646
150	2.527	2.527	2.527	149.932	-0.067718
165	2.527	2.527	2.527	164.638	-0.362366
180	2.527	2.527	2.527	178.783	-1.21729
195	2.527	2.527	2.527	193.224	-1.7756
210	2.527	2.527	2.527	207.422	-2.57785
225	2.527	2.527	2.527	221.21	-3.79041
240	2.527	2.527	2.527	235.025	-4.97545
255	2.527	2.527	2.527	249.13	-5.87018
270	2.527	2.527	2.527	267.94	-2.06
285	2.527	2.527	2.527	282.874	-2.12555
300	2.527	2.527	2.527	299.46	-0.54007
315	2.527	2.527	2.527	315.549	0.548523
330	2.527	2.527	2.527	331.649	1.64865
345	2.527	2.527	2.527	346.536	1.53625

Average Error: 2.23032

Table C.6 -- Elevation = -45°, Roll = 0°

az	biasX	biasY	biasZ	estAz	error
0	2.527	2.527	2.527	1.0501	1.0501
15	2.527	2.527	2.527	17.7735	2.77351
30	2.527	2.527	2.527	33.173	3.17304
45	2.527	2.527	2.527	47.7869	2.78693
60	2.527	2.527	2.527	62.301	2.30103
75	2.527	2.527	2.527	77.2442	2.24425
90	2.527	2.527	2.527	96.0813	6.08128
105	2.527	2.527	2.527	109.88	4.88046
120	2.527	2.527	2.527	125.225	5.22534
135	2.527	2.527	2.527	139.202	4.20184
150	2.527	2.527	2.527	153.852	3.85184
165	2.527	2.527	2.527	168.259	3.25923
180	2.527	2.527	2.527	182.627	2.6272
195	2.527	2.527	2.527	196.289	1.28906
210	2.527	2.527	2.527	211.203	1.20285
225	2.527	2.527	2.527	224.044	-0.956024
240	2.527	2.527	2.527	238.173	-1.82668
255	2.527	2.527	2.527	252.696	-2.30403
270	2.527	2.527	2.527	266.454	-3.54568
285	2.527	2.527	2.527	281.94	-3.06
300	2.527	2.527	2.527	297.295	-2.70508
315	2.527	2.527	2.527	313.11	-1.88995
330	2.527	2.527	2.527	329.405	-0.595001
345	2.527	2.527	2.527	344.747	-0.253082

Average Error: 2.67015

Table C.7 -- Elevation = -60°, Roll = 0°

az	biasX	biasY	biasZ	estAz	error
0	2.527	2.527	2.527	3.91658	3.91658
15	2.527	2.527	2.527	19.3985	4.39847
30	2.527	2.527	2.527	36.1691	6.16908
45	2.527	2.527	2.527	51.1309	6.13087
60	2.527	2.527	2.527	65.5918	5.5918
75	2.527	2.527	2.527	80.2268	5.22684
90	2.527	2.527	2.527	94.6386	4.6386
105	2.527	2.527	2.527	109.184	4.18401
120	2.527	2.527	2.527	123.444	3.44404
135	2.527	2.527	2.527	137.67	2.6702
150	2.527	2.527	2.527	151.865	1.86493
165	2.527	2.527	2.527	166.219	1.21864
180	2.527	2.527	2.527	180.273	0.272522
195	2.527	2.527	2.527	194.158	-0.842422
210	2.527	2.527	2.527	208.572	-1.42755
225	2.527	2.527	2.527	222.034	-2.96555
240	2.527	2.527	2.527	236.545	-3.45494
255	2.527	2.527	2.527	249.817	-5.18269
270	2.527	2.527	2.527	267.599	-2.40067
285	2.527	2.527	2.527	282.962	-2.03796
300	2.527	2.527	2.527	298.735	-1.26544
315	2.527	2.527	2.527	314.595	-0.404785
330	2.527	2.527	2.527	331.653	1.65347
345	2.527	2.527	2.527	347.242	2.2424

Average Error: 3.06685

Table C.8 -- Elevation = 0°, Roll = 30°

az	biasX	biasY	biasZ	estAz	error
0	2.5	2.5	2.5	359.619	-0.380676
15	2.5	2.5	2.5	14.698	-0.301957
30	2.5	2.5	2.5	30.1731	0.173098
45	2.5	2.5	2.5	45.3778	0.377831
60	2.5	2.5	2.5	58.9748	-1.02524
75	2.5	2.5	2.5	74.0228	-0.977249
90	2.5	2.5	2.5	88.2744	-1.72562
105	2.5	2.5	2.5	106.525	1.52481
120	2.5	2.5	2.5	121.171	1.17126
135	2.5	2.5	2.5	135.812	0.812195
150	2.5	2.5	2.5	150.419	0.418549
165	2.5	2.5	2.5	165.833	0.833405
180	2.5	2.5	2.5	180.573	0.572739
195	2.5	2.5	2.5	195.25	0.249908
210	2.5	2.5	2.5	209.911	-0.089202
225	2.5	2.5	2.5	223.742	-1.25833
240	2.5	2.5	2.5	238.479	-1.52118
255	2.5	2.5	2.5	251.737	-3.26323
270	2.5	2.5	2.5	266.647	-3.35251
285	2.5	2.5	2.5	283.632	-1.36777
300	2.5	2.5	2.5	299.507	-0.493408
315	2.5	2.5	2.5	314.799	-0.200836
330	2.5	2.5	2.5	330.643	0.64325
345	2.5	2.5	2.5	346.11	1.10974

Average Error: 0.9935

Table C.9 -- Elevation = 0°, Roll = 45°

az	biasX	biasY	biasZ	estAz	error
0	2.5	2.5	2.5	0.321116	0.321116
15	2.5	2.5	2.5	15.298	0.297969
30	2.5	2.5	2.5	29.2553	-0.744734
45	2.5	2.5	2.5	44.3075	-0.692501
60	2.5	2.5	2.5	58.8875	-1.11253
75	2.5	2.5	2.5	73.413	-1.58698
90	2.5	2.5	2.5	91.3848	1.38483
105	2.5	2.5	2.5	106.901	1.90138
120	2.5	2.5	2.5	122.119	2.11892
135	2.5	2.5	2.5	137.873	2.8734
150	2.5	2.5	2.5	153.388	3.38799
165	2.5	2.5	2.5	169.003	4.00301
180	2.5	2.5	2.5	184.422	4.42244
195	2.5	2.5	2.5	199.288	4.28775
210	2.5	2.5	2.5	213.875	3.87515
225	2.5	2.5	2.5	227.911	2.91138
240	2.5	2.5	2.5	243.091	3.09105
255	2.5	2.5	2.5	255.802	0.802017
270	2.5	2.5	2.5	270.874	0.874359
285	2.5	2.5	2.5	287	2.00046
300	2.5	2.5	2.5	302.2	2.19989
315	2.5	2.5	2.5	317.792	2.79211
330	2.5	2.5	2.5	331.533	1.53253
345	2.5	2.5	2.5	347.064	2.06406

Average Error: 2.13661

Table C.10 -- Elevation = 0°, Roll = 60°

az	biasX	biasY	biasZ	estAz	error
0	2.5	2.5	2.5	0.713101	0.713101
15	2.5	2.5	2.5	15.373	0.372992
30	2.5	2.5	2.5	30.4885	0.488461
45	2.5	2.5	2.5	44.9066	-0.093429
60	2.5	2.5	2.5	59.5241	-0.475864
75	2.5	2.5	2.5	74.3486	-0.651436
90	2.5	2.5	2.5	89.5007	-0.499252
105	2.5	2.5	2.5	105.895	0.894508
120	2.5	2.5	2.5	121.474	1.47446
135	2.5	2.5	2.5	136.827	1.82701
150	2.5	2.5	2.5	152.856	2.85614
165	2.5	2.5	2.5	167.872	2.8721
180	2.5	2.5	2.5	183.372	3.37248
195	2.5	2.5	2.5	198.422	3.42206
210	2.5	2.5	2.5	213.333	3.33342
225	2.5	2.5	2.5	227.954	2.95441
240	2.5	2.5	2.5	241.699	1.69888
255	2.5	2.5	2.5	255.965	0.9655
270	2.5	2.5	2.5	269.858	-0.142151
285	2.5	2.5	2.5	283.865	-1.13538
300	2.5	2.5	2.5	297.903	-2.09738
315	2.5	2.5	2.5	318.422	3.42188
330	2.5	2.5	2.5	332.487	2.48669
345	2.5	2.5	2.5	346.795	1.79504
Average Error: 1.6685					

APPENDIX D. HERCULES API C++ HEADER FILES

The Hercules Articulated Body Modeling API consists of several source code files. These programs are used to store information in efficient, re-sizable lists, create vertices and polygons, combine these vertices and polygons into rigid bodies, join these rigid bodies into articulated bodies of many parts and animate these articulated bodies in a computer graphics application.

The following C++ header files encompass the entire Hercules API. The description of their use can be found in Chapter VI and Appendix E. An example application of the Hercules API can be found in Appendix F.

1. File: HGMatrix.h

```
/*******************
   HGmatrix.h -- Matrix math library
   written by Wil Frey

   This class was designed specifically to support 3D Coordinate
   transformations.  Thus, some of the methods are defined ONLY
   for a 4x4 matrix.  These methods are designated by a 4 suffix.

   This class represents vectors as (1,X) matrices.  The default
   matrix size is (1,4).

   NOTE: matrix subscripts are zero-based.
*******************/


#pragma once

#include "HGtypes.h"


#define MAX_ARRAY_DIM 4        // largest matrix supported

class HGMatrix
{
    friend HGPoint operator* (HGPoint &, HGMatrix &);

    public:

       HGMatrix(int = 4, int = 4);                          // no init
       HGMatrix(int, int, int, float[]);                  // init with array
       HGMatrix(const HGMatrix &);                          // copy
        HGMatrix operator+(const HGMatrix &) const;     // matrix + matrix
        HGMatrix operator+(const float) const;              // matrix + scalar
        HGMatrix operator-(const HGMatrix &) const;     // matrix - matrix
        HGMatrix operator-(const float) const;              // matrix - scalar
        HGMatrix operator*(const HGMatrix &) const;     // matrix * matrix
        HGMatrix operator*(const float) const;              // matrix * scalar
        HGPoint  operator*(const HGPoint &) const;     // matrix * HGPoint
        HGMatrix operator/(const float) const;              // matrix / scalar
        HGMatrix operator=(const HGMatrix &);             // matrix assignment
        float operator()(int, int) const;                   // element access
        float operator()(int) const;                    // vector element access
        void SetElement(int, int, float);           // set only element x,y
        void SetElement(int, float);           // set only element 0,y of vector
        void Set(int, float[]);                // set entire matrix with array
        HGMatrix Transpose() const;            // standard matrix transpose
        HGMatrix Inverse() const;              // generic matrix inverse
        HGMatrix InverseH4() const;            // 4x4 special inverse matrix
        HGMatrix RotationMatrix4() const;          // 3x3 rotation matrix
        HGMatrix TranslationMatrix4() const;     // 1x3 translation matrix
        HGMatrix PerspectiveMatrix4() const;     // 1x3 perspective matrix
        void Clear();                                        // clear to identity
        int RowsQ() const;                                // rowsize of matrix
        int ColsQ() const;                                // colsize of matrix
        float *GetMatrixDataStructure() const {return &element[0][0];}

    protected:

        float       element[MAX_ARRAY_DIM][MAX_ARRAY_DIM];
        int        rows, cols;            // matrix size (rows x cols)

};
```

2. File: HGPoint.h

```
/****************
    HGPoint.h -- Vertex class for use with HG
    written by Wil Frey
*****************/


#pragma once

struct HGPoint
{
    float x, y, z, h;
    HGPoint     *next;
    HGPoint     *prev;

    HGPoint();
    HGPoint(float[]);
    HGPoint( HGPoint &);
    HGPoint &operator= ( HGPoint &);
    void SetPoint(float []);
    ~HGPoint();
};
```

3. File: HGPoly.h

```
/******************
   HGpoly.h -- Polygon class
   written by Wil Frey
*******************/

#pragma once

#include "HGtypes.h"
#include "HGArrayList.cpp"


class HGPoly
{
    friend class HGPolyList;

    public:

        HGPoly();
        HGPoly( HGPoly &); // copy
        HGPoly &operator=( HGPoly &);
        int AddVertex(int);
        void SetVertexList(int, int[]); // set vertex list from array
        int *GetVertex(int) ; // get numbered vertex
        int GetNumVertices() ;

    private:

        HGArrayList<int>        vertexList;

        HGPoly                  *next;
        HGPoly                  *prev;

};
```

4. File: HGRigidBody.h

```
/******************
   HGrigidBody.h -- Rigid Body class
   Written for Mac OS by Wil Frey
   Modified for OpenGL by Wil Frey
******************/

#pragma once

#include <math.h>
#include "HGtypes.h"
#include "HGmatrix.h"
#include "HGpoly.h"
#include "HGArrayList.cpp"

class HGRigidBody
{
    public:

        HGRigidBody();

        int AddVertex(HGPoint *); // returns vertexnum
        int AddPolygon(HGPoly *); // returns polynum
        void AppendRigidBody(HGRigidBody *); // append another object to this

        void SetPosture(HGState6f *); // set posture and calcs hMatrix
        void SetPosture(float, float, float, float, float, float);
        void SetAttachmentPoint(HGRigidBody *, int, // attach to another body
                                PositType = absolute,
                                HGRotAxisDesignator = noAxis,
                                HGRotAxisDesignator = noAxis,
                                HGRotAxisDesignator = noAxis);
        void Detatch(); // detatch body from root body
        void SetBodyMaterialType(HGMaterialType *); // set material with mat'l
        void SetBodyMaterialType(float []); // set material with array
        void SetBodyDrawingMode(DrawingModeType);
        void SetUpdateMethod(UpdateMethodType);

        virtual void UpdatePosture();
        void Transform(); // builds body hMatrix

        void ShiftPivotPoint(int); // make point x the articulation pivot
        void ShiftPivotPoint(float,float,float);// make new articulation pivot
        void Reflect(HGAxisDesignator); // reflect object along specified axis

        HGPoint *GetVertex(int) ; // get specified vertex
        HGPoint *GetTVertex(int); // get specified transformed vertex
        HGPoly *GetPoly(int) ; // get numbered poly
        HGState6f *GetPosture() {return &posture;}
        HGMatrix *GetHMatrix() {return &hMatrix;}
        HGMaterialType *GetBodyMaterialType() {return &bodyMaterial;}
        DrawingModeType GetBodyDrawingMode() {return drawingMode;}
        int GetNumVertices() ;
        int GetNumPolys() ;

        HGPoint &GetVertexNormal(int); // call after polys defined
    HGPoint &GetPolyNormal(int); // call after polys defined
    HGPoint &GetTPolyNormal(int); // call only after Transform() done

    ~HGRigidBody() {}

protected:

    // The following are articulation parameters.  They detail which
    // body this object is attached to, and at which of its points
    // to attach.  If the articulatedQ flag is not zero, then there
    // is a body to which this is attached and the position portions of
    // this object's posture are to be ignored.  If articulatedQ is zero,
    // then this object is not articulated and the position portions
    // of this object's posture are used for positioning the object.

    int                 articulatedQ;  // is this body articulated?
    HGRigidBody         *attachedTo;   // to what is this body attached?
    int                 atPointNum;    // at which point of other body?
    PositType           positRelation; // how to position joined body?
    short           azJoint;          // boolean azimuth joint angle flag
    short           elJoint;          // boolean elev joint angle flag
    short           rlJoint;          // boolean roll joint angle flag

    HGPoint             scratchPoint; // scratch data

    UpdateMethodType        updateMethod;  // posture update method
    DrawingModeType         drawingMode; // body rendering mode

    HGState6f               posture;
    HGMatrix                hMatrix;
    HGArrayList<HGPoint>    vertexList;
    HGArrayList<HGPoint>    vertexNormalList;
    HGArrayList<HGPoly>     polygonList;
    HGPoint                 rootPoint;
    HGMaterialType          bodyMaterial;

    virtual float AzimuthScript();    // override for scripted motion
    virtual float ElevationScript();
    virtual float RollScript();
    virtual float XPositScript();
    virtual float YPositScript();
    virtual float ZPositScript();

    virtual float UpdateAzimuth();    // override for physical motion
    virtual float UpdateElevation();
    virtual float UpdateRoll();
    virtual float UpdateXPosit();
    virtual float UpdateYPosit();
    virtual float UpdateZPosit();

    virtual float SensorAzimuth();    // override for sensor motion
    virtual float SensorElevation();
    virtual float SensorRoll();
    virtual float SensorXPosit();
    virtual float SensorYPosit();
    virtual float SensorZPosit();

};
```

5. File: HGViewPoint.h

```
/******************
   HGViewPoint.h -- ViewPoint class
   written by Wil Frey
*******************/


#pragma once

#include <math.h>
#include <GL/gl.h>          // Get the OpenGL required includes.
#include <GL/glu.h>
#include <GL/glx.h>

#include "HGtypes.h"
#include "HGArrayList.cpp"
#include "HGmatrix.h"
#include "HGrigidBody.h"


class HGViewPoint
{
    public:

        HGViewPoint(float enlarge=30, float nearClip=1.0,
                        float farClip=100000.0, float fov=45);

        void PositionViewPoint(HGState6f *); // sets posture and calcs hMatrix
        void RenderObject(HGRigidBody *); // renders rigid body

    protected:

        HGState6f    posture;
        float        enlargement;
        float        nearClipPlane;
        float        farClipPlane;
        float        fieldOfView;

        void turnOnTheLightModel();
      void turnOnTheLights();
      void turnOnMaterial(GLenum, HGRigidBody *);
};
```

6. File: HGPrimitives.h

```
/***************
    HGPrimitives.h -- Primitives for use with HG
    written by Wil Frey

*****************/

#pragma once

#include "HGtypes.h"
#include "HGrigidBody.h"


class HGBlock : public HGRigidBody
{
    public:
        HGBlock(float xdim=1.0, float ydim=1.0, float zdim=1.0);
};

class HGCylinder : public HGRigidBody
{
    public:
        HGCylinder(int segments=10, int makeTopFlag=1, int makeBottomFlag=1,
                    float topXPlusRadius=1.0, float topXMinusRadius=1.0,
                    float topZPlusRadius=1.0, float topZMinusRadius=1.0,
                    float bottomXPlusRadius=1.0, float bottomXMinusRadius=1.0,
                    float bottomZPlusRadius=1.0, float bottomZMinusRadius=1.0,
                    float bottomXPosit=0.0, float bottomYPosit=-1.0,
                    float bottomZPosit=0.0);
};

// generic spheroid object
// note: all radii are positive.  Negative radii allows concavity.
class HGSpheroid : public HGRigidBody
{
    public:
        HGSpheroid(int segments=10,
                    float xPlusradius=1.0, float xMinusRadius=1.0,
                    float yPlusradius=1.0, float yMinusRadius=1.0,
                    float zPlusradius=1.0, float zMinusRadius=1.0);
};
```

7. File: HGTypes.h

```
/******************
   HGtypes.h -- Type definitions
   written by Wil Frey
******************/


#pragma once

#include <math.h>
#include "HGpoint.h"

typedef struct {float az, el, rl, xp, yp, zp;} HGState6f;

typedef struct {float az, el, rl;} HGOrientation3f;

typedef struct {float xp, yp, zp;} HGPosition3f;

typedef struct {unsigned int red, green, blue, alpha;} HGRGBAColor;

typedef struct {float ambient[4];
                float diffuse[4];
                float specular[4];
                float shininess[1];} HGMaterialType;

enum DrawingModeType        {invisible,      // don't draw body
                             wireFrame,
                             opaque,
                             flatShaded,
                             smoothShaded};

enum UpdateMethodType    {noMotionUpdate,  // manual posture update only
                          scriptAnimated,  // posture updated with script
                          physicallyBased, // posture updated as physical body
                          sensorUpdated};  // update posture with sensor data

enum HGAxisDesignator    {xAxis, yAxis, zAxis};    // axis desig parameter

enum HGRotAxisDesignator   {noAxis, azimuth, elevation, roll};

enum PositType  {absolute, relative}; // articulated body positioning type

const float Pi = 3.141592653589793;
```

APPENDIX E. HERCULES API DOCUMENTATION

The author has given the Hercules API a great deal of flexibility in the construction and manipulation of both rigid and articulated bodies. This section describes each functional class of the Hercules system in detail. An experienced C++ programmer will be able to easily utilize the Hercules API to build even the most complicated articulated bodies. An example application of the API to build and perambulate a 15-segment human body is included in Appendix E. Only the applicable sections of the main C++ code body are included to show the use of the **HumanBody** class.

1. HGArrayList

The author created the HGArrayList class template to provide an expandable list structure with the approximate data retrieval speed of an array. Typically link list data structures are slow in retrieving data but are expandable in size during runtime. Arrays are not expandable once declared but are lightning fast at data retrieval. The HGArrayList class provides the AP with the best of both worlds.

AddMember() : This method is used to insert a new member into the declared HGArrayList class object. The input parameter is a pointer to a data object of the type the instantiated ArrayList is declared to hold. This function returns the list index of new member.

GetMember() : This method is used to access a particular list member. The input parameter is the list index of the desired list member. A pointer is returned which points to the desired data object.

GetMemberCount() : This method returns the size of the ArrayList as an integer count of the members currently in the list.

ClearList() : This method clears the ArrayList, releasing all storage and resetting all parameters to the initial list state.

2. HGMatrix

The author created the HGMatrix class to support the Hercules API with specific matrix operations that were required for manipulation of data structures within the API. Applications programmers are allowed access to any of the HGMatrix class member functions to allow for custom manipulation of user data structures. The class has been expanded to support most common matrix manipulations involved in three-dimensional graphics transforms.

HGMatrix(int, int) : This constructor allows the instantiation of a raw matrix class object. The first input parameter specifies the number of rows in the matrix. The second parameter specifies the number of columns. if either parameter is not specified, it defaults to 4. If the matrix is defined as square, the matrix is initialized to a unit matrix. Otherwise, all of the matrix elements are zeroed. If either input parameter is one, a row vector is assumed.

HGMatrix(int, int, int, float[]) : This constructor allows the initialization of a matrix with a specified array. The first two parameters specify the number of rows and columns, respectively. The third parameter is the number of elements in the initialization array. The fourth parameter is the array to use for initialization. The matrix size initialization constraints are the same as described above.

operator+(HGMatrix) : This method performs matrix addition between two like sized matrices only. If the matrices are not like sized, a base-initialized matrix is returned.

operator+(float) : This method performs scalar addition of the input parameter to every member of the target matrix.

operator-(HGMatrix) : This method performs matrix subtraction of the parameter matrix from the target matrix only if the matrices are like sized. If they are not he same size, a base-initialized matrix is returned.

operator-(float) : This method performs scalar subtraction of the input parameter from every member of the target matrix.

operator*(HGMatrix) : This method performs matrix post-multiplication of the target matrix by the parameter matrix. If the two matrices are not compatible for multiplication, a base-initialized matrix results.

operator*(float) : This method performs scalar multiplication of the input parameter and every member of the target matrix.

operator*(HGPoint) : This method performs vector post-multiplication of the target matrix. The resultant HGPoint has been corrected to return its fourth parameter (h) to unity for support of three-dimensional graphics transforms.

operator/(float) : This method performs scalar division of all members of the target matrix, provided division-by-zero is not attempted.

operator=(HGMatrix) : This method performs assignment of the target matrix's elements with the corresponding elements from the parameter matrix. If the parameter matrix is smaller than the target matrix, only those elements in target matrix which correspond to the parameter matrix are altered.

operator()(int, int) : This method allows the AP to have access to individual matrix elements using the parentheses operator.

operator()(int) : This method allows the AP to have access to individual vector (single-row matrix) elements using the parentheses operator.

SetElement(int, int, float) : This method allows the AP to set individual elements of the matrix to the input float parameter.

SetElement(int, float) : This method allows the AP to set individual elements of the vector to the input float parameter.

Set(int, float[]) : This method allows the AP to set the entire matrix with an input array.

Transpose() : This method performs the standard matrix transpose operation on the target matrix. The resulting matrix is returned without altering the target matrix.

Inverse() : This method calculates the standard matrix inverse of the target matrix using Gausian elimination. The resulting matrix is returned without altering the target matrix.

InverseH4() : This method performs matrix inversion of a homogeneous transformation matrix. It is specific to computer graphics transforms, and thus has a more simple inverse than a generic matrix.

RotationMatrix4() : This method returns a 3x3 matrix which is the rotation portion of a homogeneous transformation matrix.

TranslationMatrix4() : This method returns a 1x3 matrix which is the translation portion of a homogeneous transformation matrix.

PerspectiveMatrix4() : This method returns a 1x3 matrix which is the perspective portion of a homogeneous transformation matrix.

Clear() : This method clears the matrix to the unity matrix, if square, and zeroes otherwise.

RowsQ() : Returns the number of rows in the matrix.

ColsQ() : Returns the number of columns in the matrix.

GetMatrixDataStructure() : Returns the matrix as a reference to the two-dimensional array data structure in which the matrix elements are stored. This function is very useful for sending the contents of the matrix to SGI OpenGL routines.

3. HGPoint

This class (structure) was created to allow flexibility in manipulation of vectors representing various aspects of the Hercules API, especially RigidBody vertices.

HGPoint(float[]) : This constructor initializes the point using the input array.

operator=(HGPoint) : This method allows the point to be easily copied from another HGPoint.

SetPoint(float[]) : This method allows the elements of a point to be set using the input array.

4. HGPoly

This class was developed by the author to allow ease of polygon definition and manipulation. This class simplifies the Hercules internal data structures.

operator=(HGPoly) : This method allows the polygon to be duplicated from the input polygon.

AddVertex(int) : This method adds the numbered RigidBody vertex to the vertex list. In actuality, only the index of the vertex is added. Later, Hercules uses this index to access the vertex itself. The vertex is never specifically referenced by the HGPoly class.

SetVertexList(int, int[]) : This method is used to build the polygon's vertex list from the specified integer array.

GetVertex(int) : This method returns the a pointer to the index of the specified vertex.

GetNumVertices() : Returns the number of vertices which define the polygon.

5. HGRigidBody

The HGRigidBody class is the workhorse of the entire Hercules API. It contains the rigid body definition (vertices and polygons), its attachment information, material specification, drawing mode and update method. It allows the rigid body to be manipulated by shifting its pivot point, reflecting it through a plane, positioning and orienting it in space and appending it to another rigid body.

AddVertex(HGPoint *) : This method adds the specified vertex to the rigid body's vertex list.

AddPolygon(HGPoly *) : This method adds the specified polygon to the rigid body's polygon list. Addition of a polygon also updates the vertex normals of the vertices that the polygon uses. This is done to allow proper Gouraud shading of the rigid body.

AppendRigidBody(HGRigidBody *) : This method appends the entire specified rigid body to the target rigid body object. The specified body's vertex list, vertex normal list and polygon list are all copied into the target body's lists. The specified body takes on the material and drawing mode characteristics of the target body.

SetPosture(HGState6f *) : This method sets the rigid body's position and orientation using the specified HGState6f data structure (see HGTypes.h for the structure definition).

SetPosture(float, float, float, float, float, float) : This method sets the rigid body's position and orientation using the six input parameters. In order, the input parameters are azimuth, elevation, roll, x position, y position, z position.

SetAttachmentPoint() : This method is used to set the parent of the target rigid body. The parent is the body to which the target rigid body will remain attached to throughout its manipulations. The parameters of this function are a pointer to the parent HGRigidbody to which the target body is to be attached, the vertex of the parent body at which to attach the target body, the method of orientation (either absolute or relative) and the axes of rotation about which motion is allowed under the relative orientation method.

Detatch() : This method detaches the target body from its parent.

SetBodyMaterialType() : This method sets the body OpenGL material type using the specified HGMaterialType data structure (see HGTypes.h for the structure definition).

SetBodyMaterialType(float []) : This method sets the body OpenGL material type using the specified 13 element array (see HGTypes.h for the structure definition).

SetBodyDrawingMode(DrawingModeType) : This method sets the body drawing mode to either wireFrame, flatShaded or smoothShaded. wireFrame drawing mode does not remove hidden surfaces. smoothShaded uses the standard Gouraud shading algorithm included in OpenGL.

SetUpdateMethod(UpdateMethodType) : This method allows the AP to set the rigid body's automatic posture updating method (scriptAnimated, physicallyBased, sensorUpdated). This parameter is used to select the proper update method when the UpdatePosture method is called.

UpdatePosture() : This method allows the AP a standard facility for updating the posture of the rigid body. The possible selections for update method are scriptAnimated, physicallyBased and sensorUpdated. It is up to the AP to provide the specific posture updating methods for any inherited class of HGRigidBody which intends to use the UpdatePosture method. The scriptAnimated methods are called AzimuthScript, ElevationScript, RollScript, XPositScript, YPositScript and ZPositScript. The physicallyBased methods are called UpdateAzimuth, UpdateElevation, UpdateRoll, UpdateXPosit, UpdateYPosit and UpdateZPosit. The sensorUpdated methods are not supported yet.

Transform() : This method is called by the AP to caused the graphical transformation of the rigid body according to the body's posture. This method causes the body's H-matrix to be built and allows the object to be rendered properly. If the Transform method is not called before the object is rendered, the object will be drawn with its last transformed posture.

ShiftPivotPoint(int) : This method shifts all of the body's vertices so that the new local origin of the body is placed at the specified vertex. In other words, the body will subsequently rotate around the specified vertex.

ShiftPivotPoint(float,float,float) : This method also shifts the pivot point of the body, but pivot point is shifted to the specified local coordinates.

Reflect(HGAxisDesignator) : This method reflects all of the body's vertices through a plane perpendicular to the specified axis (xAxis, yAxis or zAxis). This is useful for making bodies which are mirror images of each other.

GetVertex(int) : This method returns a pointer to the specified non-transformed vertex.

GetTVertex(int) : This method returns a pointer to the specified transformed vertex (the global coordinates of the vertex at the current body posture).

GetPoly(int) : This method returns a pointer to the specified polygon.

GetPosture() : This method returns a pointer to the body's current posture data structure.

GetHMatrix() : This method returns a pointer to the body's current H-Matrix.

GetBodyMaterialType() : This method returns a pointer to the body's material type data structure.

GetBodyDrawingMode() : This method returns the body's drawing mode.

GetNumVertices() : This method returns the number of vertices contained in the body's vertex list.

GetNumPolys() : This method returns the number of polygons contained in the body's polygon list.

GetVertexNormal(int) : This method returns a reference to the specified vertex's normal vector. This is necessary for Gouraud shading in OpenGL.

GetPolyNormal(int) : This method returns a reference to the specified polygon's normal vector. This is necessary for flat shading in OpenGL.

GetTPolyNormal(int) : This method returns a reference to the specified polygon normal vector in global coordinates.

6. HGViewPoint

The HGViewPoint class is the code responsible for generating a view in the virtual world. Each ViewPoint object has its own posture, representing its position and view direction in the world. A simple lighting model has been incorporated into the HGViewPoint class. It is not modifiable. That functionality will be added in later version of the HGViewPoint code.

HGViewPoint(float enlarge, float nearClip, float farClip, float fov) : This constructor allows the AP to create the viewpoint object and specify the enlargement factor, near clipping plane, far clipping plane and field-of-view. These are the same parameters as defined for OpenGL drawing environments. [OPENGL 94]

PositionViewPoint(HGState6f *) : This method allows the AP to position and orient the viewpoint.

RenderObject(HGRigidBody *) : This method renders the specified rigid body from the current viewpoint's frame of reference.

7. HGPrimitives

HGPrimitives classes are inherited from the base HGRigidBody classes and represent three basic, pre-prepared building blocks: HGBlock, HGCylinder and HGSpheroid. These primitives can be used to build more complicated rigid body structures by using the HGRigidBody method AppendRigidBody.

HGBlock(float xdim, float ydim, float zdim) : This primitive is used to make a right rectangular block with x, y, and z dimensions of xdim, ydim and zdim, respectively.

```
HGCylinder(int segments=10, int makeTopFlag=1, int makeBottomFlag=1,
           float topXPlusRadius=1.0, float topXMinusRadius=1.0,
           float topZPlusRadius=1.0, float topZMinusRadius=1.0,
           float bottomXPlusRadius=1.0, float bottomXMinusRadius=1.0,
           float bottomZPlusRadius=1.0, float bottomZMinusRadius=1.0,
           float bottomXPosit=0.0, float bottomYPosit=-1.0,
           float bottomZPosit=0.0)
```

The HGCylinder primitive is a fairly complicated class to use because of the number of parameters passed. However, the class is very flexible in the variety of cylinders which can be made. Note that the cylinder is formed in the upright position with the y-axis being along its cylindrical axis.

By giving the PlusRadius and MinusRadius parameters different values, a cylinder with different shaped opposing sides can be made. By making the X- and Z-radius parameters different, a squashed cylinder can be made. Making the top and bottom radii different will form a tapered cylinder. By setting bottomXPosit and / or bottomZPosit to values other than zero, a skewed cylinder can be made. Setting the bottomYPosit parameter controls the height of the cylinder. The segments parameter controls how many segments the walls of the cylinder are broken up into. The makeTopFlag and makeBottomFlag parameters define to the routine whether or not the polygons for the top and the bottom of the cylinders should be generated.

```
HGSpheroid(int  segments=10,
           float xPlusradius=1.0, float xMinusRadius=1.0,
           float yPlusradius=1.0, float yMinusRadius=1.0,
           float zPlusradius=1.0, float zMinusRadius=1.0)
```

The HGSpheroid primitive allows the creation of various types of spheroidal shapes. By setting the radius parameters to different values, various flattened and extended spheroidal shapes can be formed. Setting both axis radii to either a positive or negative value will allow the generation of a spheroid with a concave (rather than convex) side. The segments parameter controls how many longitudinal and latitudinal segments the spheroid is broken up into.

APPENDIX F. EXAMPLE HERCULES APPLICATION

The following classes are used to build and manipulate a fifteen-segment human body using the Hercules system. This is just one example. The author has also constructed and animated a 27-segment articulated Black Widow Spider. The same basic techniques are used for both. As the human model is useful in more virtual environment applications, it is included here instead of the spider.

The animation routines are designed to give the human model a life-like walking or running motion, depending on the rate parameter passed to the motion routine. However, no physical basis was used for the walk script. Only sine and cosine functions were used to give an appropriate period appearance to the script.

1. File: HumanBodyParts.h

```
/***************
    HumanBodyParts.h -- Example application using Herc API
    written by Wil Frey
*****************/

#include "HGtypes.h"
#include "HGrigidBody.h"

enum Command {stand, walk, halt};

float DegToRad(float);


// Human Torso
class HumanTorso : public HGRigidBody
{
    public:
        HumanTorso();
        void UpdatePosture(Command, float, float, float);
};
```

```
// Human Head
class HumanHead : public HGRigidBody
{
    public:
        HumanHead();
        void UpdatePosture(Command, float, float, float);
};

// Human Hips
class HumanHips : public HGRigidBody
{
    public:
        HumanHips();
        void UpdatePosture(Command, float, float, float);
};

// Human Upper Legs
class HumanUpperLeg : public HGRigidBody
{
    public:
        HumanUpperLeg();
};

class HumanRULeg : public HumanUpperLeg
{
    public:
        HumanRULeg() {}
        void UpdatePosture(Command, float, float, float);
};

class HumanLULeg : public HumanUpperLeg
{
    public:
        HumanLULeg() {}
        void UpdatePosture(Command, float, float, float);
};

// Human Lower Legs
class HumanLowerLeg : public HGRigidBody
{
    public:
        HumanLowerLeg();
};

class HumanRLLeg : public HumanLowerLeg
{
    public:
        HumanRLLeg() {}
        void UpdatePosture(Command, float, float, float);
};

class HumanLLLeg : public HumanLowerLeg
{
    public:
        HumanLLLeg() {}
        void UpdatePosture(Command, float, float, float);
};
```

```
//Human Feet
class HumanFoot : public HGRigidBody
{
    public:
        HumanFoot();
};

class HumanRFoot : public HumanFoot
{
    public:
        HumanRFoot() {}
        void UpdatePosture(Command, float, float, float);
};

class HumanLFoot : public HumanFoot
{
    public:
        HumanLFoot() {}
        void UpdatePosture(Command, float, float, float);
};

//Human Upper Arms
class HumanRUArm : public HGRigidBody
{
    public:
        HumanRUArm();
        void UpdatePosture(Command, float, float, float);
};

class HumanLUArm : public HGRigidBody
{
    public:
        HumanLUArm();
        void UpdatePosture(Command, float, float, float);
};

//Human Lower Arms
class HumanLowerArm : public HGRigidBody
{
    public:
        HumanLowerArm();
};

class HumanRLArm : public HumanLowerArm
{
    public:
        HumanRLArm() {}
        void UpdatePosture(Command, float, float, float);
};

class HumanLLArm : public HumanLowerArm
{
    public:
        HumanLLArm() {}
        void UpdatePosture(Command, float, float, float);
};
```

```
//Human Hands
class HumanRHand : public HGRigidBody
{
    public:
        HumanRHand();
        void UpdatePosture(Command, float, float, float);
};

class HumanLHand : public HGRigidBody
{
    public:
        HumanLHand();
        void UpdatePosture(Command, float, float, float);
};
```

2. File: HumanBodyParts.cpp

```
/***************
    HumanBodyParts.cpp -- Example application using Herc API
    written by Wil Frey
****************/

#include "HumanBodyParts.h"


float DegToRad(float inDegrees)
{
    return (inDegrees / 180) * Pi;
}


HumanTorso::HumanTorso()
{
    int index;
    HGPoint    point;
    HGPoly    poly;

    float    pArray[] = {0,0,0,1,        // 0 -- root

                        2,-1.5,-2.5,1,    // 1 -- front
                        -2,-1.5,-2.5,1,    // 2
                        2,1,-1,1,        // 3
                        -2,1,-1,1,        // 4
                        4,0.5,-1,1,        // 5
                        -4,0.5,-1,1,    // 6
                        4,-1,-1,1,        // 7
                        -4,-1,-1,1,        // 8
                        3,-2,-1,1,        // 9
                        -3,-2,-1,1,        // 10
                        2,-7,-1,1,        // 11
                        -2,-7,-1,1,        // 12

                        2.5,0,2,1,        // 13 -- back
                        -2.5,0,2,1,        // 14
                        2,1,1,1,        // 15
                        -2,1,1,1,        // 16
                        4,0.5,1,1,        // 17
                        -4,0.5,1,1,        // 18
                        4,-1,1,1,        // 19
                        -4,-1,1,1,        // 20
                        3,-2,1,1,        // 21
                        -3,-2,1,1,        // 22
                        2,-7,1,1,        // 23
                        -2,-7,1,1,        // 24

                        0,-6.75,0,1,    // 25 -- hips anchor
                        0,1,0,1,        // 26 -- head anchor
                        4,0,0,1,        // 27 -- right arm anchor
                        -4,0,0,1};        // 28 -- left arm anchor
    for (index = 0; index < 116; index += 4)
    {
        point.SetPoint(&pArray[index]);
        AddVertex(&point);
    }

    int     vertex1[] =  {1,2,4,3,            // front quad's
                          11,12,2,1,
                          13,15,16,14,        // back quad's
                          13,14,24,23,
                          3,4,16,15,         // lateral quad's
                          4,6,18,16,
                          6,8,20,18,
                          8,10,22,20,
                          10,12,24,22,
                          12,11,23,24,
                          11,9,21,23,
                          9,7,19,21,
                          7,5,17,19,
                          5,3,15,17};

    for(index = 0; index < 56; index+=4)
    {
        poly.SetVertexList(4, &vertex1[index]);
        AddPolygon(&poly);
    }

    int     vertex2[] =   {1,3,5,          // front tri's
                          1,5,7,
                          1,7,9,
                          1,9,11,
                          2,12,10,
                          2,10,8,
                          2,8,6,
                          2,6,4,
                          14,16,18,     // back tri's
                          14,18,20,
                          14,20,22,
                          14,22,24,
                          13,23,21,
                          13,21,19,
                          13,19,17,
                          13,17,15};

    for(index = 0; index < 48; index+=3)
    {
        poly.SetVertexList(3, &vertex2[index]);
        AddPolygon(&poly);
    }

    SetBodyDrawingMode(flatShaded);

    SetUpdateMethod(scriptAnimated);

    return;
}
```

```
void HumanTorso::UpdatePosture(Command com, float dir, float rate, float timestamp)
{
    float deltaMove, constantMove, rateStamp, lastRateStamp;
    static float lastTimeStamp = 0;

    switch (com)
    {
        case walk:
            rateStamp = rate * timestamp;
            lastRateStamp = rate * lastTimeStamp;

            posture.az = dir + (DegToRad(20) * -sin(rateStamp));
            posture.el = -DegToRad(rate/2);
            posture.rl = 0;

            constantMove = (rateStamp - lastRateStamp) * rate * rate / 40;
            deltaMove = (14 - constantMove)
                        * fabs( fabs( sin( 0.5236 * sin(rateStamp)))
                        - fabs( sin( 0.5236 * sin(lastRateStamp))))
                        + constantMove;

            posture.xp += deltaMove * -sin(dir);
            posture.zp += deltaMove * -cos(dir);

            posture.yp = (rate / 20) * (1 - (fabs(cos(rateStamp))
                         * fabs(cos(rateStamp))));

            break;

        case stand:
        default:
            posture.az = dir;
            posture.el = 0;
            posture.rl = 0;
    }

    Transform();

    lastTimeStamp = timestamp;

    return;
}
```

```
//**********************************************************
HumanHead::HumanHead()
{
    int index;
    HGPoint    point;
    HGPoly     poly;

    float    pArray[] = {0,0,0,1,                 // 0 -- root

                         1,0,-1,1,                // 1
                         -1,0,-1,1,               // 2
                         1,0.5,-1,1,              // 3
                         -1,0.5,-1,1,             // 4
                         1.25,2.75,-1.45,1,       // 5
                         -1.25,2.75,-1.45,1,      // 6
                         1.15,3.5,-1.35,1,        // 7
                         -1.15,3.5,-1.35,1,       // 8

                         1,0,1,1,                 // 1
                         -1,0,1,1,                // 2
                         1,0.5,1,1,               // 3
                         -1,0.5,1,1,              // 4
                         1.25,2.75,1.45,1,        // 5
                         -1.25,2.75,1.45,1,       // 6
                         1.15,3.5,1.35,1,         // 7
                         -1.15,3.5,1.35,1,        // 16

                         0.75,2.3,-2.6,1,         // 17 -- hat bill
                         -0.75,2.3,-2.6,1,        // 18

                         0,2.25,-1.25,1,          // 19 -- nose
                         0.3,1.5,-1.1,1,          // 20
                         -0.3,1.5,-1.1,1,         // 21
                         0,1.4,-1.6,1};           // 22

    for (index = 0; index < 92; index += 4)
    {
        point.SetPoint(&pArray[index]);
        AddVertex(&point);
    }

    int    vertex[] =   {1,2,4,3,        // front quad's
                        3,4,6,5,
                        5,6,8,7,

                        9,11,12,10,     // back quad's
                        11,13,14,12,
                        13,15,16,14,

                        7,8,16,15,      // side quad's
                        8,6,14,16,
                        6,4,12,14,
                        4,2,10,12,
                        2,1,9,10,
                        1,3,11,9,
                        3,5,13,11,
                        5,7,15,13,
                        5,17,18,6,      // hat bill
                        5,6,18,17,

                        19,20,22,19,     // nose
                        19,22,21,19,
                        20,21,22,20};

    for(index = 0; index < 76; index+=4)
    {
        poly.SetVertexList(4, &vertex[index]);
        AddPolygon(&poly);
    }

    SetUpdateMethod(scriptAnimated);

    return;
}


void HumanHead::UpdatePosture(Command com, float dir, float rate, float timestamp)
{
    switch (com)
    {
        case walk:
            posture.az = dir;
            posture.el = DegToRad(2.5) * sin(2 * rate * timestamp);
            posture.rl = 0;
            break;

        case stand:
        default:
            posture.az = dir;
            posture.el = 0;
            posture.rl = 0;
    }

    Transform();

    return;
}
```

```
//*************************************************************
HumanHips::HumanHips()
{
    int index;
    HGPoint    point;
    HGPoly    poly;

    float    pArray[] = {0,0,0,1,           // 0 -- root

                         2,0,-1,1,           // 1 -- front
                         -2,0,-1,1,           // 2
                         2.5,-1.5,-1.25,1,    // 3
                         -2.5,-1.5,-1.25,1,    // 4
                         2.5,-3,-1,1,        // 5
                         -2.5,-3,-1,1,        // 6

                         2,0,1,1,            // 7 -- back
                         -2,0,1,1,            // 8
                         2.5,-1.5,1.25,1,    // 9
                         -2.5,-1.5,1.25,1,    // 10
                         2.5,-3,1,1,        // 11
                         -2.5,-3,1,1,        // 12

                         1.5,-2.75,0,1,        // 13 -- right leg anchor
                         -1.5,-2.75,0,1};    // 14 -- left leg anchor

    for (index = 0; index < 60; index += 4)
    {
        point.SetPoint(&pArray[index]);
        AddVertex(&point);
    }

    int    vertex[] =    {1,3,4,2,           // front quad's
                       3,5,6,4,
                       7,8,10,9,          // back quad's
                       9,10,12,11,
                       1,2,8,7,          // side quad's
                       2,4,10,8,
                       4,6,12,10,
                       6,5,11,12,
                       5,3,9,11,
                       3,1,7,9};

    for(index = 0; index < 40; index+=4)
    {
        poly.SetVertexList(4, &vertex[index]);
        AddPolygon(&poly);
    }

    SetUpdateMethod(scriptAnimated);

    return;
}
```

```
void HumanHips::UpdatePosture(Command com, float dir, float rate, float timestamp)
{
    switch (com)
    {
        case walk:
            posture.az = dir + (DegToRad(10) * sin(rate * timestamp));
            posture.el = 0;
            posture.rl = DegToRad(10) * -sin(rate * timestamp);
            break;

        case stand:
        default:
            posture.az = dir;
            posture.el = 0;
            posture.rl = 0;
    }

    Transform();

    return;
}
```

```
//*************************************************************
HumanUpperLeg::HumanUpperLeg()
{
    int index;
    HGPoint    point;
    HGPoly    poly;

    float    pArray[] = {0,0,0,1,            // root

                         1,0,-1,1,            // 1 -- front
                         -1,0,-1,1,            // 2
                         1.25,-2,-1.25,1,    // 3
                         -1.25,-2,-1.25,1,    // 4
                         0.75,-6.5,-0.75,1,    // 5
                         -0.75,-6.5,-0.75,1,    // 6

                         1,0,1,1,            // 7 -- front
                         -1,0,1,1,            // 8
                         1.25,-2,1.25,1,        // 9
                         -1.25,-2,1.25,1,    // 10
                         0.75,-6.5,0.75,1,    // 11
                         -0.75,-6.5,0.75,1,    // 12

                         0,-6.25,0,1};        // 13 -- lower leg anchor

    for (index = 0; index < 56; index += 4)
    {
        point.SetPoint(&pArray[index]);
        AddVertex(&point);
    }

    int    vertex[] = {1,3,4,2,        // front quad's
                     3,5,6,4,

                     7,8,10,9,        // back quad's
                     9,10,12,11,

                     1,2,8,7,        // side quad's
                     2,4,10,8,
                     4,6,12,10,
                     6,5,11,12,
                     5,3,9,11,
                     3,1,7,9};

    for(index = 0; index < 40; index+=4)
    {
        poly.SetVertexList(4, &vertex[index]);
        AddPolygon(&poly);
    }

    SetUpdateMethod(scriptAnimated);

    return;
}
```

```
// Human right upper leg scripts
void HumanRULeg::UpdatePosture(Command com, float dir, float rate, float timestamp)
{
    switch (com)
    {
        case walk:
            posture.az = dir;
            posture.el = DegToRad(30) * sin(rate * timestamp);
            posture.rl = DegToRad(3) * -fabs(sin(rate * timestamp));
            break;

        case stand:
        default:
            posture.az = dir;
            posture.el = 0;
            posture.rl = 0;
    }

    Transform();

    return;
}


// Human left upper leg scripts
void HumanLULeg::UpdatePosture(Command com, float dir, float rate, float timestamp)
{
    switch (com)
    {
        case walk:
            posture.az = dir;
            posture.el = DegToRad(30) * -sin(rate * timestamp);
            posture.rl = DegToRad(3) * fabs(sin(rate * timestamp));
            break;

        case stand:
        default:
            posture.az = dir;
            posture.el = 0;
            posture.rl = 0;
    }

    Transform();

    return;
}
```

```
//***********************************************************
HumanLowerLeg::HumanLowerLeg()
{
    int index;
    HGPoint    point;
    HGPoly    poly;

    float    pArray[] = {0,0,0,1,           // root

                        0.75,0,-0.75,1,       // 1 -- front
                        -0.75,0,-0.75,1,    // 2
                        1,-2,-1,1,           // 3
                        -1,-2,-1,1,           // 4
                        0.25,-6.5,-0.5,1,    // 5
                        -0.25,-6.5,-0.5,1,    // 6

                        0.75,0,0.75,1,       // 7 -- front
                        -0.75,0,0.75,1,       // 8
                        1,-2,1,1,            // 9
                        -1,-2,1,1,            // 10
                        0.25,-6.5,0.5,1,    // 11
                        -0.25,-6.5,0.5,1,    // 12

                        0,-6.25,0,1};        // 13 -- foot anchor

    for (index = 0; index < 56; index += 4)
    {
        point.SetPoint(&pArray[index]);
        AddVertex(&point);
    }

    int    vertex[] = {1,3,4,2,        // front quad's
                    3,5,6,4,

                    7,8,10,9,        // back quad's
                    9,10,12,11,

                    1,2,8,7,        // side quad's
                    2,4,10,8,
                    4,6,12,10,
                    6,5,11,12,
                    5,3,9,11,
                    3,1,7,9};

    for(index = 0; index < 40; index+=4)
    {
        poly.SetVertexList(4, &vertex[index]);
        AddPolygon(&poly);
    }

    SetUpdateMethod(scriptAnimated);

    return;
}
```

```
// Human right lower leg scripts
void HumanRLLeg::UpdatePosture(Command com, float dir, float rate, float timestamp)
{
    float animPoint = fmod((rate * timestamp),(2 * Pi));

    switch (com)
    {
        case walk:
            posture.az = dir;
            if (animPoint > (3 * Pi / 2) || animPoint < (Pi / 2))
                posture.el = (DegToRad(30) * sin(rate * timestamp)) -
                                (DegToRad(7*rate) * cos(rate * timestamp));
            else
                posture.el = DegToRad(30) * sin(rate * timestamp);
            posture.rl = 0;
            break;

        case stand:
        default:
            posture.az = dir;
            posture.el = 0;
            posture.rl = 0;
    }

    Transform();

    return;
}
```

```
// Human left lower leg scripts
void HumanLLLeg::UpdatePosture(Command com, float dir, float rate, float timestamp)
{
    float animPoint = fmod((rate * timestamp),(2 * Pi));

    switch (com)
    {
        case walk:
            posture.az = dir;
            if (animPoint < (3 * Pi / 2) && animPoint > (Pi / 2))
                posture.el = (DegToRad(7*rate) * cos(rate * timestamp)) -
                                 (DegToRad(30) * sin(rate * timestamp));
            else
                posture.el = DegToRad(30) * -sin(rate * timestamp);
            posture.rl = 0;
            break;

        case stand:
        default:
            posture.az = dir;
            posture.el = 0;
            posture.rl = 0;
    }

    Transform();

    return;
}
```

```
//**************************************************************
HumanFoot::HumanFoot()
{
    int index;
    HGPoint    point;
    HGPoly    poly;

    float    pArray[] = {0,0,0,1,                // root

                      -0.25,0,-0.5,1,             // 1 -- left side
                      -0.25,0,0.5,1,             // 2
                      -0.25,-1.25,-1,1,         // 3
                      -0.5,-1.75,1,1,             // 4
                      -0.25,-2,0.75,1,         // 5
                      -0.5,-2,-2.25,1,         // 6
                      -0.25,-2,-2.75,1,         // 7
                      -0.25,-1.75,-3,1,         // 8
                      -0.75,-1.75,-2.5,1,         // 9

                      0.25,0,-0.5,1,             // 10 -- right side
                      0.25,0,0.5,1,             // 11
                      0.25,-1.25,-1,1,         // 12
                      0.5,-1.75,1,1,             // 13
                      0.25,-2,0.75,1,             // 14
                      0.5,-2,-2.25,1,             // 15
                      0.25,-2,-2.75,1,         // 16
                      0.25,-1.75,-3,1,         // 17
                      0.75,-1.75,-2.5,1};         // 18

    for (index = 0; index < 76; index += 4)
    {
        point.SetPoint(&pArray[index]);
        AddVertex(&point);
    }

    int    vertex1[] = {1,3,2,        // foot tri's
                    2,3,4,
                    3,9,4,
                    10,11,12,
                    11,13,12,
                    12,13,18};

    for(index = 0; index < 18; index+=3)
    {
        poly.SetVertexList(3, &vertex1[index]);
        AddPolygon(&poly);
    }
    int     vertex2[] = {4,9,6,5,          // foot quads's
                         9,8,7,6,
                         13,14,15,18,
                         15,16,17,18,
                         1,2,11,10,
                         2,4,13,11,
                         4,5,14,13,
                         5,6,15,14,
                         6,7,16,15,
                         7,8,17,16,
                         8,9,18,17,
                         9,3,12,18,
                         3,1,10,12};

    for(index = 0; index < 52; index+=4)
    {
        poly.SetVertexList(4, &vertex2[index]);
        AddPolygon(&poly);
    }

    SetUpdateMethod(scriptAnimated);

    return;
}


// Human right foot scripts
void HumanRFoot::UpdatePosture(Command com, float dir, float rate, float timestamp)
{
    float animPoint = fmod((rate * timestamp),(2 * Pi));

    switch (com)
    {
        case walk:
            posture.az = dir;
            if (animPoint > (3 * Pi / 2) || animPoint < (Pi / 2))
                posture.el = (DegToRad(30) * sin(rate * timestamp)) -
                             (DegToRad(7*rate) * cos(rate * timestamp));
            else
                posture.el = DegToRad(30) * sin(rate * timestamp);
            posture.rl = 0;
            break;

        case stand:
        default:
            posture.az = dir;
            posture.el = 0;
            posture.rl = 0;
    }

    Transform();

    return;
}
```

```
// Human left foot scripts
void HumanLFoot::UpdatePosture(Command com, float dir, float rate, float timestamp)
{
    float animPoint = fmod((rate * timestamp),(2 * Pi));

    switch (com)
    {
        case walk:
            posture.az = dir;
            if (animPoint < (3 * Pi / 2) && animPoint > (Pi / 2))
                posture.el = (DegToRad(7*rate) * cos(rate * timestamp)) -
                                    (DegToRad(30) * sin(rate * timestamp));
            else
                posture.el = DegToRad(30) * -sin(rate * timestamp);
            posture.rl = 0;
            break;

        case stand:
        default:
            posture.az = dir;
            posture.el = 0;
            posture.rl = 0;
    }

    Transform();

    return;
}


//*********************************************************
HumanRUArm::HumanRUArm()
{
    int index;
    HGPoint    point;
    HGPoly    poly;

    float    pArray[] = {0,0,0,1,                  // root

                        1,-0.25,-1,1,              // 1 -- front
                        0,0.5,-1,1,                  // 2
                        1.25,-2.25,-1,1,           // 3
                        -0.25,-2.25,-1,1,           // 4
                        1,-5,-0.5,1,               // 5
                        0,-5,-0.5,1,               // 6

                        1,-0.25,1,1,               // 7 -- front
                        0,0.5,1,1,                   // 8
                        1.25,-2.25,1,1,              // 9
                        -0.25,-2.25,1,1,           // 10
                        1,-5,0.5,1,                  // 11
                        0,-5,0.5,1,                  // 12

                        0.5,-4.75,0,1};              // 13 -- R lower arm anchor
    for (index = 0; index < 56; index += 4)
    {
        point.SetPoint(&pArray[index]);
        AddVertex(&point);
    }

    int    vertex[] = {1,3,4,2,         // front quad's
                       3,5,6,4,

                       7,8,10,9,        // back quad's
                       9,10,12,11,

                       1,2,8,7,         // side quad's
                       2,4,10,8,
                       4,6,12,10,
                       6,5,11,12,
                       5,3,9,11,
                       3,1,7,9};

    for(index = 0; index < 40; index+=4)
    {
        poly.SetVertexList(4, &vertex[index]);
        AddPolygon(&poly);
    }

    SetUpdateMethod(scriptAnimated);

    return;
}


void HumanRUArm::UpdatePosture(Command com, float dir, float rate, float timestamp)
{
    switch (com)
    {
        case walk:
            posture.az = dir;
            posture.el = DegToRad(30) * -sin(rate * timestamp);
            posture.rl = 0;
            break;

        case stand:
        default:
            posture.az = dir;
            posture.el = 0;
            posture.rl = 0;
    }

    Transform();

    return;
}
```

```
//*************************************************************
HumanLUArm::HumanLUArm()
{
    int index;
    HGPoint    point;
    HGPoly    poly;

    float    pArray[] = {0,0,0,1,                // root

                      0,0.5,-1,1,                 // 1 -- front
                      -1,-0.25,-1,1,             // 2
                      0.25,-2.25,-1,1,          // 3
                      -1.25,-2.25,-1,1,         // 4
                      0,-5,-0.5,1,              // 5
                      -1,-5,-0.5,1,              // 6

                      0,0.5,1,1,                  // 7 -- front
                      -1,-0.25,1,1,              // 8
                      0.25,-2.25,1,1,              // 9
                      -1.25,-2.25,1,1,          // 10
                      0,-5,0.5,1,                  // 11
                      -1,-5,0.5,1,              // 12

                      -0.5,-4.75,0,1};          // 13 -- L lower arm anchor

    for (index = 0; index < 56; index += 4)
    {
        point.SetPoint(&pArray[index]);
        AddVertex(&point);
    }

    int    vertex[] = {1,3,4,2,         // front quad's
                   3,5,6,4,

                   7,8,10,9,         // back quad's
                   9,10,12,11,

                   1,2,8,7,         // side quad's
                   2,4,10,8,
                   4,6,12,10,
                   6,5,11,12,
                   5,3,9,11,
                   3,1,7,9};

    for(index = 0; index < 40; index+=4)
    {
        poly.SetVertexList(4, &vertex[index]);
        AddPolygon(&poly);
    }

    SetUpdateMethod(scriptAnimated);

    return;
}
```

```
void HumanLUArm::UpdatePosture(Command com, float dir, float rate, float timestamp)
{
    switch (com)
    {
        case walk:
            posture.az = dir;
            posture.el = DegToRad(30) * sin(rate * timestamp);
            posture.rl = 0;
            break;

        case stand:
        default:
            posture.az = dir;
            posture.el = 0;
            posture.rl = 0;
    }

    Transform();

    return;
}


//*****************************************************************
HumanLowerArm::HumanLowerArm()
{
    int index;
    HGPoint    point;
    HGPoly    poly;

    float    pArray[] = {0,0,0,1,                  // root

                        0.5,0,-0.5,1,              // 1 -- front
                        -0.5,0,-0.5,1,              // 2
                        0.75,-1.5,-0.75,1,          // 3
                        -0.75,-1.5,-0.75,1,          // 4
                        0.25,-5,-0.3,1,              // 5
                        -0.25,-5,-0.3,1,           // 6

                        0.5,0,0.5,1,               // 7 -- back
                        -0.5,0,0.5,1,               // 8
                        0.75,-1.5,0.75,1,           // 9
                        -0.75,-1.5,0.75,1,           // 10
                        0.25,-5,0.3,1,               // 11
                        -0.25,-5,0.3,1,               // 12

                        0,-4.75,0,1};               // 13 -- hand anchor

    for (index = 0; index < 56; index += 4)
    {
        point.SetPoint(&pArray[index]);
        AddVertex(&point);
    }
    int     vertex[] = {1,3,4,2,          // front quad's
                        3,5,6,4,

                        7,8,10,9,         // back quad's
                        9,10,12,11,

                        1,2,8,7,          // side quad's
                        2,4,10,8,
                        4,6,12,10,
                        6,5,11,12,
                        5,3,9,11,
                        3,1,7,9};

    for(index = 0; index < 40; index+=4)
    {
        poly.SetVertexList(4, &vertex[index]);
        AddPolygon(&poly);
    }

    SetUpdateMethod(scriptAnimated);

    return;
}


// Human right lower arm script
void HumanRLArm::UpdatePosture(Command com, float dir, float rate, float timestamp)
{
    switch (com)
    {
        case walk:
            posture.az = dir;
            posture.el = DegToRad(4*rate) - DegToRad(30) * sin(rate * timestamp);
            posture.rl = 0;
            break;

        case stand:
        default:
            posture.az = dir;
            posture.el = 0;
            posture.rl = 0;
    }

    Transform();

    return;
}
```

```
// Human left lower arm script
void HumanLLArm::UpdatePosture(Command com, float dir, float rate, float timestamp)
{
    switch (com)
    {
        case walk:
            posture.az = dir;
            posture.el = DegToRad(4*rate) + DegToRad(30) * sin(rate * timestamp);
            posture.rl = 0;
            break;

        case stand:
        default:
            posture.az = dir;
            posture.el = 0;
            posture.rl = 0;
    }

    Transform();

    return;
}


//*************************************************************
HumanRHand::HumanRHand()
{
    int index;
    HGPoint    point;
    HGPoly    poly;

    float    pArray[] = {0,0,0,1,                  // root

                      -0.25,0,-0.15,1,          // 1 -- palm side
                      -0.25,0,0.3,1,              // 2
                      -0.25,-1.5,-0.3,1,          // 3
                      -0.25,-1.5,0.75,1,          // 4
                      -0.25,-3,-0.15,1,          // 5
                      -0.25,-3,0.3,1,              // 6
                      -0.25,0,-0.3,1,              // 7
                      -0.25,-0.5,-0.75,1,          // 8
                      -0.25,-2,-0.6,1,          // 9
                      -0.25,-2,-0.3,1,          // 10

                      0.25,0,-0.15,1,              // 11 -- back side
                      0.25,0,0.3,1,              // 12
                      0.4,-1.5,-0.3,1,          // 13
                      0.4,-1.5,0.75,1,          // 14
                      0,-3,-0.15,1,              // 15
                      0,-3,0.3,1,                  // 16
                      0.25,0,-0.3,1,              // 17
                      0.25,-0.5,-0.75,1,          // 18
                      0,-2,-0.6,1,              // 19
                      0,-2,-0.3,1,              // 20
                      0.1,-1.5,-0.3,1};          // 21
    for (index = 0; index < 88; index += 4)
    {
        point.SetPoint(&pArray[index]);
        AddVertex(&point);
    }

    int    vertex[] = {1,3,4,2,        // back side quad's
                       3,5,6,4,
                       1,7,8,3,
                       3,8,9,10,

                       11,12,14,13,    // palm side quad's
                       13,14,16,15,
                       11,13,18,17,
                       13,20,19,18,

                       2,12,17,7,
                       7,17,18,8,
                       8,18,19,9,
                       9,19,20,10,
                       10,20,21,3,
                       3,13,15,5,
                       5,15,16,6,
                       6,16,14,4,
                       4,14,12,2};

    for(index = 0; index < 68; index+=4)
    {
        poly.SetVertexList(4, &vertex[index]);
        AddPolygon(&poly);
    }

    SetUpdateMethod(scriptAnimated);

    return;
}
```

```
// Human right hand script
void HumanRHand::UpdatePosture(Command com, float dir, float rate, float timestamp)
{
    switch (com)
    {
        case walk:
            posture.az = dir;
            posture.el = DegToRad(4*rate+15) - DegToRad(30) * sin(rate * timestamp);
            posture.rl = 0;
            break;

        case stand:
        default:
            posture.az = dir;
            posture.el = 0;
            posture.rl = 0;
    }

    Transform();

    return;
}


//*****************************************************************
HumanLHand::HumanLHand()
{
    int index;
    HGPoint    point;
    HGPoly    poly;

    float    pArray[] = {0,0,0,1,                // root

                       0.25,0,-0.15,1,           // 1 -- palm side
                       0.25,0,0.3,1,             // 2
                       0.25,-1.5,-0.3,1,         // 3
                       0.25,-1.5,0.75,1,         // 4
                       0.25,-3,-0.15,1,          // 5
                       0.25,-3,0.3,1,            // 6
                       0.25,0,-0.3,1,            // 7
                       0.25,-0.5,-0.75,1,        // 8
                       0.25,-2,-0.6,1,           // 9
                       0.25,-2,-0.3,1,           // 10

                       -0.25,0,-0.15,1,          // 11 -- back side
                       -0.25,0,0.3,1,            // 12
                       -0.4,-1.5,-0.3,1,         // 13
                       -0.4,-1.5,0.75,1,         // 14
                       0,-3,-0.15,1,             // 15
                       0,-3,0.3,1,               // 16
                       -0.25,0,-0.3,1,           // 17
                       -0.25,-0.5,-0.75,1,       // 18
                       0,-2,-0.6,1,              // 19
                       0,-2,-0.3,1,              // 20
                       -0.1,-1.5,-0.3,1};        // 21
    for (index = 0; index < 88; index += 4)
    {
        point.SetPoint(&pArray[index]);
        AddVertex(&point);
    }

    int     vertex[] = {1,2,4,3,         // back side quad's
                        3,4,6,5,
                        1,3,8,7,
                        3,10,9,8,

                        11,13,14,12,     // palm side quad's
                        13,15,16,14,
                        11,17,18,13,
                        13,18,19,20,

                        2,7,17,12,
                        7,8,18,17,
                        8,9,19,18,
                        9,10,20,19,
                        10,3,21,20,
                        3,5,15,13,
                        5,6,16,15,
                        6,4,14,16,
                        4,2,12,14};

    for(index = 0; index < 68; index+=4)
    {
        poly.SetVertexList(4, &vertex[index]);
        AddPolygon(&poly);
    }

    SetUpdateMethod(scriptAnimated);

    return;
}
```

```
// Human left hand script
void HumanLHand::UpdatePosture(Command com, float dir, float rate, float timestamp)
{
    switch (com)
    {
        case walk:
            posture.az = dir;
            posture.el = DegToRad(4*rate+15) + DegToRad(30) * sin(rate * timestamp);
            posture.rl = 0;
            break;

        case stand:
        default:
            posture.az = dir;
            posture.el = 0;
            posture.rl = 0;
    }

    Transform();

    return;
}
```

3. File: HumanBody.h

```
/**************
    HumanBody.h -- overall human body object
***************/

#pragma once

#include "HumanBodyParts.h"
#include "HGViewPoint.h"

// Human Body
class HumanBody
{
    public:

        HumanTorso          torso;
        HumanHead           head;
        HumanHips           hips;
        HumanRULeg          ruleg;
        HumanLULeg          luleg;
        HumanRLLeg          rlleg;
        HumanLLLeg          llleg;
        HumanRFoot          rfoot;
        HumanLFoot          lfoot;
        HumanRUArm          ruarm;
        HumanLUArm          luarm;
        HumanRLArm          rlarm;
        HumanLLArm          llarm;
        HumanRHand          rhand;
        HumanLHand          lhand;

        Command currentMotion;
        Command commandedMotion;

        float currentDirection;
        float commandedDirection;

        float currentRate;
        float commandedRate;

        float currentTimeStamp;

        HumanBody();
        void SetBodyDrawingMode(DrawingModeType);
        void SetBodyMaterialType(HGMaterialType *);
        void Motion(Command com, float direction, float rate, float timeStamp);
        void Render(HGViewPoint *viewPoint);
};
```

4. File: HumanBody.cpp

```
/**************
    HumanBody.cpp -- overall human body object
    written for MacOS by Wil Frey
    modified for OpenGL by Wil Frey
***************/

#pragma once

#include "HumanBody.h"


// overall human body
HumanBody::HumanBody()
{
    head.SetAttachmentPoint(&torso, 26, absolute);
    hips.SetAttachmentPoint(&torso, 25, absolute);
    ruleg.SetAttachmentPoint(&hips, 13, absolute);
    luleg.SetAttachmentPoint(&hips, 14, absolute);
    rlleg.SetAttachmentPoint(&ruleg, 13, absolute);
    llleg.SetAttachmentPoint(&luleg, 13, absolute);
    rfoot.SetAttachmentPoint(&rlleg, 13, absolute);
    lfoot.SetAttachmentPoint(&llleg, 13, absolute);
    ruarm.SetAttachmentPoint(&torso, 27, absolute);
    luarm.SetAttachmentPoint(&torso, 28, absolute);
    rlarm.SetAttachmentPoint(&ruarm, 13, absolute);
    llarm.SetAttachmentPoint(&luarm, 13, absolute);
    rhand.SetAttachmentPoint(&rlarm, 13, absolute);
    lhand.SetAttachmentPoint(&llarm, 13, absolute);

    SetBodyDrawingMode(wireFrame);

    currentTimeStamp = 0;

    currentDirection = 0;
    commandedDirection = 0;

    currentMotion = stand;
    commandedMotion = stand;

    return;
}
```

```
void HumanBody::SetBodyDrawingMode(DrawingModeType mode)
{
    torso.SetBodyDrawingMode(mode);
    head.SetBodyDrawingMode(mode);
    hips.SetBodyDrawingMode(mode);
    ruleg.SetBodyDrawingMode(mode);
    luleg.SetBodyDrawingMode(mode);
    rlleg.SetBodyDrawingMode(mode);
    llleg.SetBodyDrawingMode(mode);
    rfoot.SetBodyDrawingMode(mode);
    lfoot.SetBodyDrawingMode(mode);
    ruarm.SetBodyDrawingMode(mode);
    luarm.SetBodyDrawingMode(mode);
    rlarm.SetBodyDrawingMode(mode);
    llarm.SetBodyDrawingMode(mode);
    rhand.SetBodyDrawingMode(mode);
    lhand.SetBodyDrawingMode(mode);

    return;
}


void HumanBody::SetBodyMaterialType(HGMaterialType *material)
{
    torso.SetBodyMaterialType(material);
    head.SetBodyMaterialType(material);
    hips.SetBodyMaterialType(material);
    ruleg.SetBodyMaterialType(material);
    luleg.SetBodyMaterialType(material);
    rlleg.SetBodyMaterialType(material);
    llleg.SetBodyMaterialType(material);
    rfoot.SetBodyMaterialType(material);
    lfoot.SetBodyMaterialType(material);
    ruarm.SetBodyMaterialType(material);
    luarm.SetBodyMaterialType(material);
    rlarm.SetBodyMaterialType(material);
    llarm.SetBodyMaterialType(material);
    rhand.SetBodyMaterialType(material);
    lhand.SetBodyMaterialType(material);

    return;
}
```

```
void HumanBody::Motion(Command com, float direction, float rate, float timeStamp)
{
    Command passedCom;

    switch (com)
    {
        case stand:
            switch (currentMotion)
            {
                case stand:
                case halt:
                default:
                    passedCom = stand;
                    currentTimeStamp = 0;
                    break;

                case walk:
                    if ((Pi - fmod((currentRate * currentTimeStamp),Pi)) > (Pi /
20))
                    {
                        passedCom = walk;
                        currentDirection = direction;
                        currentTimeStamp = timeStamp;
                    }

                    else
                    {
                        currentMotion = stand;
                        currentRate = 0;
                        currentDirection = direction;
                        currentTimeStamp = 0;
                        passedCom = stand;
                    }
                    break;
            }
            break;

        case walk:
            switch (currentMotion)
            {
                case stand:
                    currentMotion = walk;
                    passedCom = walk;
                    currentRate = rate;
                    currentDirection = direction;
                    currentTimeStamp = timeStamp;
                    break;

                case walk:
                    passedCom = walk;
                    currentDirection = direction;
                    currentTimeStamp = timeStamp;
                    break;
            }
            break;

    }
    torso.UpdatePosture(passedCom, currentDirection, currentRate, currentTimeStamp);
    head.UpdatePosture(passedCom, currentDirection, currentRate, currentTimeStamp);
    hips.UpdatePosture(passedCom, currentDirection, currentRate, currentTimeStamp);
    ruleg.UpdatePosture(passedCom, currentDirection, currentRate, currentTimeStamp);
    luleg.UpdatePosture(passedCom, currentDirection, currentRate, currentTimeStamp);
    rlleg.UpdatePosture(passedCom, currentDirection, currentRate, currentTimeStamp);
    llleg.UpdatePosture(passedCom, currentDirection, currentRate, currentTimeStamp);
    rfoot.UpdatePosture(passedCom, currentDirection, currentRate, currentTimeStamp);
    lfoot.UpdatePosture(passedCom, currentDirection, currentRate, currentTimeStamp);
    ruarm.UpdatePosture(passedCom, currentDirection, currentRate, currentTimeStamp);
    luarm.UpdatePosture(passedCom, currentDirection, currentRate, currentTimeStamp);
    rlarm.UpdatePosture(passedCom, currentDirection, currentRate, currentTimeStamp);
    llarm.UpdatePosture(passedCom, currentDirection, currentRate, currentTimeStamp);
    rhand.UpdatePosture(passedCom, currentDirection, currentRate, currentTimeStamp);
    lhand.UpdatePosture(passedCom, currentDirection, currentRate, currentTimeStamp);

    return;
}


void HumanBody::Render(HGViewPoint *viewPoint)
{
    viewPoint->RenderObject(&torso);
    viewPoint->RenderObject(&head);
    viewPoint->RenderObject(&hips);
    viewPoint->RenderObject(&ruleg);
    viewPoint->RenderObject(&luleg);
    viewPoint->RenderObject(&rlleg);
    viewPoint->RenderObject(&llleg);
    viewPoint->RenderObject(&rfoot);
    viewPoint->RenderObject(&lfoot);
    viewPoint->RenderObject(&ruarm);
    viewPoint->RenderObject(&luarm);
    viewPoint->RenderObject(&rlarm);
    viewPoint->RenderObject(&llarm);
    viewPoint->RenderObject(&rhand);
    viewPoint->RenderObject(&lhand);

    return;
}
```

LIST OF REFERENCES

[BACH96] Bachman, E. and Gay, D., 1996, *Design and Evaluation of an Integrated GPS/INS System for Shallow-Water AUV Navigation (SANS),* Master's Thesis, Computer Sciencs Department, Naval Postgraduate School, Monterey, California.

[BADL93] Badler, N., 1993, *Simulating Humans: Computer Graphics Animation and Control,* Oxford University Press, New York, NY.

[BIBL95] Bible, S., 1995, *Using Spread-Spectrum Ranging Techniques for Position Tracking in a Virtual Environment,* Computer Science Department, Naval Postgraduate School, Monterey, California.

[BOWD77] Bowditch, N., 1977, *American Practical Navigator,* United States Defense Mapping Agency Hydrographic Center.

[BROW92] Brown, R. and Hwang, P., 1992, *Introduction to Random Signals and Applied Kalman Filtering, Second Edition,* John Wiley and Sons, Inc., New York, NY.

[BROX64] Broxmeyer, C., 1964, *Inertial Navigation Systems,* McGraw-Hill Book Company, New York, New York.

[COOK92] Cooke, J., 1992, *NPSNET: Flight Simulation Dynamic Modeling Using Quaternions,* "Presence: Teleoperators and Virtual Environments", Volume 1, Number 4, Fall 1992.

[CRAI89] Craig, J., 1989, *Introduction to Robotics, Mechanics and Control, Second Edition*, Addison-Wesley Publishing Company, Menlo Park, California.

[DEVE86] Develco, 1986, *Operation Manual, 9200 Series 3-Axis Fluxgate Magnetometer,* Develco, 175 Nortech Parkway, San Jose, California, 95134-2306.

[FOXL94] Foxlin, E. and Durlach, N., 1994, *An Inertial Head-Orientation Tracker with Automatic Drift Compensation for Use with HMD's*, Massachusetts Institute of Technology, Cambridge, Massachusetts.

[FREY95] Frey, W., 1995, *Off-the-Shelf, Real-Time, Human Body Motion Capture for Synthetic Environments*, Computer Science Department, Naval Postgraduate School, Monterey, California.

[HOWA66] Howard, I. and Templeton, W., 1966, *Human Spatial Orientation* , John Wiley and Sons Ltd., London, England.

[MCGH93] McGhee, R., 1993, *CS-4314 Class Notes: Derivation of Body Angular Rates to Euler Angle Rates Relationship,* Computer Science Department, Naval Postgraduate School, Monterey, California.

[MCGH95] McGhee, R., 1995, *An Experimental Study of An Integrated GPS/INS System for Shallow-Water AUV Navigation (SANS),* Proceedings of the 9th International Symposium on Unmanned, Untethered Submersible Technology, Durham, NH.

[MCGH96A] McGhee, R., 1996, *CS-4920 Class Notes: Derivation of SANS Filter Equations,* Computer Science Department, Naval Postgraduate School, Monterey, California.

[MCGH96B] McGhee, R., 1996, *Research Notes: Estimation of Heading From a 3-Axis Magnetometer,* Computer Science Department, Naval Postgraduate School, Monterey, California.

[MCMI96] McMillan, S., 1996, *Upper Body Tracking Using the Polhemus Fastrak,* Technical Report NPSCS-96-002, Naval Postgraduate School, Monterey, California.

[NRC95] National Research Council (NRC), Committee on Virtual Reality Research and Development, 1995, *Virtual Reality : Scientific and Technological Challenges*, NationalAcademyPress, Washington, DC.

[ODON64] O'Donnell, C., 1964, *Inertial Navigation Analysis and Design,* McGraw-Hill Book Company, New York, New York.

[OPEN94] OpenGL Architecture Review Board, 1994, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release 1,* Addison-Wesley Publishing Company, Reading, Massachusetts.

[SKOP96] Skopowski, P., 1996, *Human Upper Body Motion Tracking: A Kinematic Approach*, Computer Science Department, Naval Postgraduate School, Monterey, California.

[WALD95] Waldrop, M., 1995, *Real-Time Articulation of the Upper Body for Simulated Humans in Virtual Environments,* Master's Thesis, Computer Science Department, Naval Postgraduate School, Monterey, California.

BIBLIOGRAPHY

Fisher, S., 1992, *Virtual Interface Environments*, "Presence: Teleoperators and Virtual Environments", Volume 1, Number 1, Winter 1992.

Hollands, R., 1995, *Sourceless Trackers*, "VR News", Volume 4, Issue 3, April 1995.

Meyer, K. and Applewhite, H., 1992, *A Survey of Position Trackers*, "Presence: Teleoperators and Virtual Environments", Volume 1, Number 2, Spring 1992.

Robinett, W., 1992, *Synthetic Experience: A Proposed Taxonomy*. "Presence: Teleoperators and Virtual Environments", Volume 1, Number 2, Spring 1992.

Rogers, D. and Adams, J., 1990, *Mathematical Elements for Computer Graphics,* McGraw-Hill, Inc., New York, NY.

Ruediger, S., 1996, *Simulation-Based Validation of Navigation Filter Software for a Shallow Water AUV Navigation System,* Masters Thesis, Computer Science Department, Naval Postgraduate School, Monterey, California.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Road, Suite 0944
   Fort Belvoir, VA 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Road
   Monterey, California 93943-5101

3. Doctor Ted Lewis 1
   Chairman
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

4. Doctor Michael Zyda 1
   Associate Chairman
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

5. Doctor Robert McGhee 1
   Professor
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

6. Russ Whalen 1
   Manager, Center for AUV Research
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

6. Eric Foxlin 1
   Research Laboratory of Electronics
   Massachusettes Institute of Technology
   Cambridge, MA 02139

7. William Frey 5
   c/o Judith Harper
   Gen. Del. Cedar Mtn. Rd.
   Divided, CO 80814