1993-12

Computer graphics tools for the visualization of spacecraft dynamics

Haynes, Keith Lorenzo.

Monterey, California. Naval Postgraduate School

(2)

NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

Computer Graphics Tools for the Visualization of Spacecraft Dynamics

By

Keith Lorenzo Haynes

December 1993

Thesis Advisor: I Michael Ross

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	15 December 1993	Master's Thesis

4. TITLE AND SUBTITLE Computer Graphics Tools for the Visualization of Spacecraft Dynamics

5. FUNDING NUMBERS

6. AUTHOR(S) Keith Lorenzo Haynes

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
*A

13. ABSTRACT *(maximum 200 words)*

This thesis consists of teaching tools designed to allow spacecraft engineering students to visualize the various phenomena associated with spacecraft dynamics. It does so via the use of state of the art three dimensional computer graphics on Silicon Graphics computers. The thesis discusses the principles in dynamics that were implemented and the key design considerations. A central goal was to develop applications that were user friendly. A library of functions were developed called Dynamics Programming Library or DPL. DPL shields the users from the details of computer graphics, thus allowing them to concentrate on the dynamics of the problem. DPL was used to write three main applications: Euler, Frame, and Gyro. Euler demonstrates the representation of orientation using Euler angles and quaternion rotation. Gyro demonstrates the effects of torques applied to varying rigid body geometries and inertias. Frame allows students to view the motion of an object from different frames of reference. A group of 21 spacecraft engineering students participated in a lab exercise using these programs. Within 20 minutes, the students could run the simulations thus validating their user friendliness.

14. SUBJECT TERMS *Type the keywords for your thesis in over these words; all keywords must be unclassified.

15. NUMBER OF PAGES
154

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Approved for public release; distribution is unlimited

Computer Graphics Tools for the Visualization of Spacecraft Dynamics

by

Keith Lorenzo Haynes
Captain, United States Army
B.A., Hofstra University, 1985
M.S., Golden Gate University, 1990

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE
MASTER OF SCIENCE IN ENGINEERING SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1993

Author: [redacted]
Keith Lorenzo Haynes

Approved By: [redacted]
I. Michael Ross, Thesis Advisor

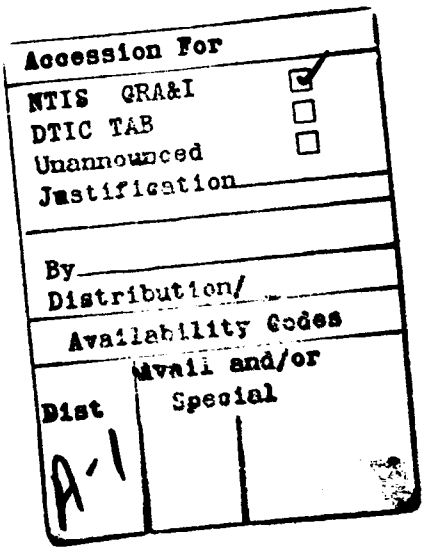
[redacted]
Robert B. McGhee, Second Reader

[redacted]
Ted Lewis, Chairman, Department of Computer Science

[redacted]
Daniel J. Collins, Chairman, Department of Aeronautics & Astronautics

ABSTRACT

This thesis consists of teaching tools designed to allow spacecraft engineering students to visualize the various phenomena associated with spacecraft dynamics. It does so via the use of state of the art three dimensional computer graphics on Silicon Graphics computers. The thesis discusses the principles in dynamics that were implemented and the key design considerations. A central goal was to develop applications that were user friendly. A library of functions were developed called Dynamics Programming Library or DPL. DPL shields the users from the details of computer graphics, thus allowing them to concentrate on the dynamics of the problem. DPL was used to write three main applications: Euler, Frame, and Gyro. Euler demonstrates the representation of orientation using Euler angles and quaternion rotation. Gyro demonstrates the effects of torques applied to varying rigid body geometries and inertias. Frame allows students to view the motion of an object from different frames of reference. A group of 21 spacecraft engineering students participated in a lab exercise using these programs. Within 20 minutes, the students could run the simulations thus validating their user friendliness.



ACKNOWLEDGMENT

I would like to thank my wife Vanessa. With her love and support, all things are possible. I would like also like to thank Professors I Michael Ross and Robert B. McGhee for their patience and guidance.

TABLE OF CONTENTS

I. INTRODUCTION

This thesis consists of teaching tools designed to allow spacecraft engineering students to visualize the various phenomena associated with spacecraft dynamics. It does so via the use of state of the art three dimensional computer graphics on Silicon Graphics computers. The system is fully interactive and incorporates the fundamental laws of dynamics.

The system consists of four main parts. The first is a Dynamics Programming Library (DPL) that allows students to write programs that graphically demonstrate particle and rigid body dynamics. The language is designed to be used by students with a background in dynamics, but does not require them to have a knowlege of computer graphics.

The other three parts of the system consist of three programs written utilizing DPL. The first program demonstrates the representation of orientation using euler angles and quaternion rotation. The second program demonstrates the effects of torques applied to varying rigid body geometries and inertias. This program allows the student to select a geometry and apply moments to the rigid body and observe the effect. The third program allows students to view the motion of an object from different frames of reference. The student is allowed to define the motion and view it from any frame of reference.

This thesis first discusses the principles in dynamics that were implemented. Next it covers the topics in computer graphics central to DPL. An important point is that users of DPL are shielded from these details. It then goes on to discuss the key design considerations of the project.

II. DYNAMICS

A. INTRODUCTION

This chapter will discuss certain key principles in dynamics and kinematics demonstrated by this thesis. It will deal with the concept of a frame of reference as well as the various methods of representing an object's orientation. Finally, particle and rigid body dynamics will be discussed.

B. FRAMES OF REFERENCES

The motion of a person driving down the highway at 55 mph can be described in different ways. Relative to the surface of the earth she is moving at 55 mph in a given direction. But relative to the car she is not moving at all. Which one is correct? Can she at the same time be moving at 55 mph and 0 mph? The answer is that they are both correct. When you consider her motion in the earth's coordinate system or frame of reference she is traveling at 55 mph. When you consider her motion in the frame of reference of the car, it is indeed 0 mph. However, the car motion in the earth's frames of reference is 55 mph. By adding the two together we get 55 mph, the same value as in the first case. The description of the motion of a body is not complete without also discussing the frame in which the motion is described.

There are an infinite number of frames that can be used to describe a body's motion. The key is to pick the frame that reveals the most about what the body is doing. A well chosen frame of reference can also simplify the interpretation of a complicated motion. There are two frames that are very useful in both regards. The first is an *inertial* frame of reference. This is a frame in which Newton's laws of motion hold. In the example of the woman driving the car, the earth would be considered an inertial frame of reference. *Body coordinates* are the other important frame of reference. This is a coordinate system fixed to a body. The location of objects fixed to the body described in body coordinates do not vary. Take for example the car and the location of the driver and passenger seats. If you are driving north in earth coordinates the driver's seat is west of the passengers; if you are

driving south, it is east of the passenger seat. However, in the car's body coordinate system the location of the driver's seat does not change. If you are driving north, the driver's seat is left of the passenger's seat. And if you are driving south it is still left of the passenger's seat.

C. ORIENTATION REPRESENTATION

When dealing with bodies that are not treated as a point mass, location of the center of mass is not sufficient information to describe what is happening to the body. A means is needed to describe the attitude or the orientation of the body. In general terms this can be thought of as describing the orientation of the body's coordinate system relative to another coordinate system. There are three primary methods of representing a coordinate system's orientation; Direction Cosine Matrix, Euler Angles, and Quaternions.

1. Direction Cosine Matrix

Given two coordinate systems A and B, each defined by a set of mutually orthogonal unit vectors, the Direction Cosine Matrix (DCM) relates the orientation of one coordinate system to the other (see Figure 2.1).

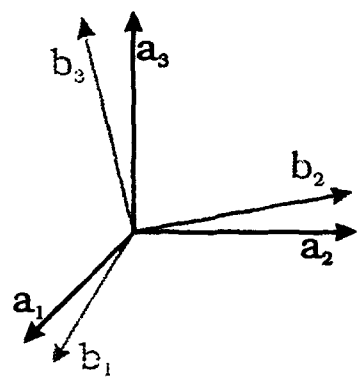


Figure 2.1 Coordinate Systems.

One interpretation of a DCM is as a matrix that can be used to change the basis of a vector from one coordinate system into another. For example, $[V]_A$ is a vector expressed in the A coordinate system.

$$[V]_A = C_{a1}\mathbf{a}_1 + C_{a2}\mathbf{a}_2 + C_{a3}\mathbf{a}_3 \tag{2.1}$$

${}^{B}R^{A}$ is the DCM that changes the basis of a vector from the **A** coordinate system to that of the **B** system. The equation below transforms $[V]_A$ into **B** coordinates.

$$[V]_B = {}^{B}R^{A}[V]_A \tag{2.2}$$

$$[V]_B = C_{b1}\mathbf{b}_1 + C_{b2}\mathbf{b}_2 + C_{b3}\mathbf{b}_3. \tag{2.3}$$

$[V]_A$ is now expressed in **B** coordinates. $[V]_B$ coincides with $[V]_A$; they are just expressed in different coordinate systems.

The question is how is ${}^{B}R^{A}$ constructed? The matrix below shows the elements in the DCM. Each element of the matrix is derived from the dot product between the two unit vectors shown. [Ref 1: p. 9] Each element is called a *direction cosine* because it is the dot product of two unit vectors.

$$[V]_B = {}^{B}R^{A}[V]_A \tag{2.4}$$

$$\begin{bmatrix} C_{b1} \\ C_{b2} \\ C_{b3} \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \cdot \mathbf{a}_1 & \mathbf{b}_1 \cdot \mathbf{a}_2 & \mathbf{b}_1 \cdot \mathbf{a}_3 \\ \mathbf{b}_2 \cdot \mathbf{a}_1 & \mathbf{b}_2 \cdot \mathbf{a}_2 & \mathbf{b}_2 \cdot \mathbf{a}_3 \\ \mathbf{b}_3 \cdot \mathbf{a}_1 & \mathbf{b}_3 \cdot \mathbf{a}_2 & \mathbf{b}_3 \cdot \mathbf{a}_3 \end{bmatrix} \begin{bmatrix} C_{a1} \\ C_{a2} \\ C_{a3} \end{bmatrix} \tag{2.5}$$

2. Euler Angle Representation

Another interpretation of the transformation matrix is a rotation about one or more of the axes of a coordinate system, see Figure 2.2. Each of these rotations can be represented by a three by three matrix. These are called the *elementary rotation* matrices and are listed on the next page. [Ref 1: p. 10]

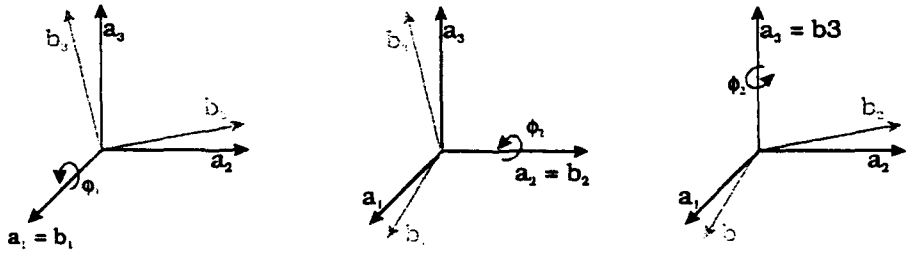


Figure 2.2 Single rotation about the **A** coordinate axes

$$ {}^{B}R^{A}(\phi_1) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi_1 & -\sin\phi_1 \\ 0 & \sin\phi_1 & \cos\phi_1 \end{bmatrix} \tag{2.6} $$

$$ {}^{B}R^{A}(\phi_2) = \begin{bmatrix} \cos\phi_2 & 0 & \sin\phi_2 \\ 0 & 1 & 0 \\ -\sin\phi_2 & 0 & \cos\phi_2 \end{bmatrix} \tag{2.7} $$

$$ {}^{B}R^{A}(\phi_3) = \begin{bmatrix} \cos\phi_3 & -\sin\phi_3 & 0 \\ \sin\phi_3 & \cos\phi_3 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.8} $$

An Euler angle rotation is a sequence of rotations about the axes of one coordinate system to eventually align them with another system. As shown in Table 2.1, there are 12 possible Euler rotation sequences: six non repeating and six repeating. They are identified by that axes about which they rotate.

Non-Repeating Sequences	Repeating Sequences
123	121
132	131
213	212
231	232
312	313
321	323

Table 2.1 Euler rotation sequences

${}^{B}R^{A}(123)$ represents the rotation matrix necessary to transform vectors defined in coordinate system **a** to **b**. ${}^{B}R^{A}(123)$ is constructed by multiplying three elementary rotation matrices. In order to transform vectors defined in **a** coordinate system to **b** multiply them by ${}^{B}R^{A}(123)$. By taking the transpose of ${}^{B}R^{A}(123)$ is the matrix to transform vectors defined in **b** coordinate system to **a**. In other words ${}^{A}R^{B}(123) = ({}^{B}R^{A}(123))^{T}$, and

$$[V]_B = {}^B R^A(123)[V]_A \tag{2.9}$$

where

$$ {}^B R^A(123) = \begin{bmatrix} \cos\phi_3 & -\sin\phi_3 & 0 \\ \sin\phi_3 & \cos\phi_3 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\phi_2 & 0 & \sin\phi_2 \\ 0 & 1 & 0 \\ -\sin\phi_2 & 0 & \cos\phi_2 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi_1 & -\sin\phi_1 \\ 0 & \sin\phi_1 & \cos\phi_1 \end{bmatrix} \tag{2.10}$$

A problem with Euler rotations is that there is a singularity in the angular rates and that the solution for every orientation is not unique. In order to clarify this point, meanings will be associated with the coordinate axes. Roll, Pitch and Heading will be used instead of 1, 2, 3 (see Figure 1.3).

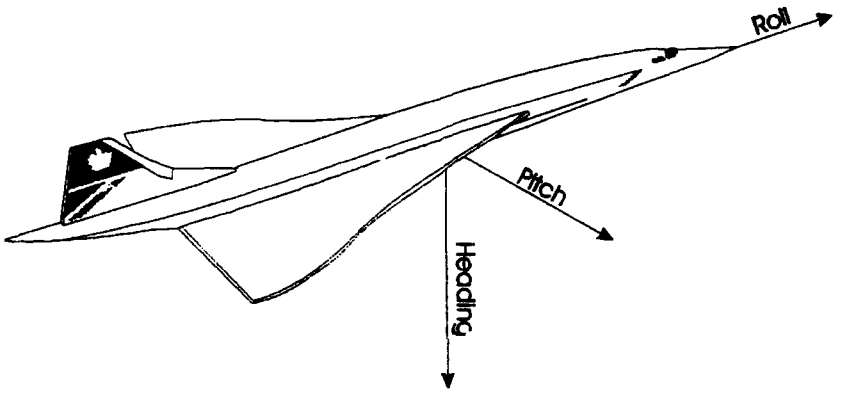


Figure 1.3 Location of Roll, Pitch, and Heading Axes.

The heading axis comes out of the belly of the aircraft and is perpendicular to the horizontal direction of flight. If the rotation about the pitch axis is 90°, which means the aircraft is flying vertically, heading no longer has meaning because there is no horizontal direction of flight. A small increase or decrease in pitch from the singularity points cause the value in heading to change rapidly. For example, if the aircraft had a heading of 0° prior to the singularity, a small decrease in pitch would return to heading to 0°, whereas, a small increase would change the heading to 180°. There are two changes like this when a plane performs a loop.

3. Quaternion Representation

Most of the time, the singularity with Euler angles is not a problem, but when it is quaternions provide an alternative representation. A quaternion Q consists of two parts: a scalar and a vector **q**. Transformation between coordinate systems is accomplished by a single rotation about the vector **q**. The scalar part of the quaternion determines the magnitude of the rotation (see Figure 1.4). [Ref 2:, pp. 9-12]

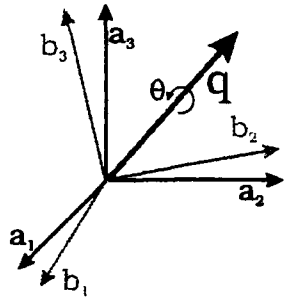


Figure 1.4 Quaternion Rotation about **q**

The quaternion Q can be written as follows:

$$Q = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix} \tag{2.11}$$

where

$q_0 = \cos(\theta/2)$ (2.12)

q_1 = (cos of angle **q** makes with x axis)$\sin(\theta/2)$ (2.13)

q_2 = (cos of angle **q** makes with y axis)$\sin(\theta/2)$ (2.14)

q_3 = (cos of angle **q** makes with z axis)$\sin(\theta/2)$ (2.15)

and θ is the rotation about **q**, $0 \leq \theta < \pi$

The quaternion can be used to generate a rotation matrix to transform vectors from one coordinate system to another. The rotation matrix is defined as follows: [Ref 3: p. 11]

$$R = \begin{bmatrix} 2(q_0^2 + q_1^2) - 1 & 2(q_1q_2 + q_0q_3) & 2(q_1q_3 - q_0q_2) \\ 2(q_1q_2 - q_0q_3) & 2(q_0^2 + q_2^2) - 1 & 2(q_2q_3 + q_0q_1) \\ 2(q_1q_3 + q_0q_2) & 2(q_2q_3 - q_0q_1) & 2(q_0^2 + q_3^2) - 1 \end{bmatrix} \quad (2.16)$$

Additionally, given a rotation matrix R, a quaternion Q can be generated using the following set of equations: [Ref 3: p. 11]

$$q_0 = 0.5(1 + r_{11} - r_{22} - r_{33})^{1/2} \quad (2.17)$$
$$q_1 = 0.5(1 - r_{11} + r_{22} - r_{33})^{1/2} \quad (2.18)$$
$$q_2 = 0.5(1 - r_{11} - r_{22} + r_{33})^{1/2} \quad (2.19)$$
$$q_3 = 0.5(1 + r_{11} + r_{22} + r_{33})^{1/2} \quad (2.20)$$

where r_{ij} are the elements of the rotation matrix.

Finally, given the angular velocity of an object in body coordinates, a simple set of equations describe the rate of change of the quaternion. These equations are shown below: [Ref 2:, pp. 9-12]

$$\dot{q}_0 = -0.5(q_1\omega_1 + q_2\omega_2 + q_3\omega_3) \quad (2.21)$$
$$\dot{q}_1 = 0.5(q_0\omega_1 + q_2\omega_3 - q_3\omega_2) \quad (2.22)$$
$$\dot{q}_2 = 0.5(q_0\omega_2 + q_3\omega_1 - q_1\omega_3) \quad (2.23)$$
$$\dot{q}_3 = 0.5(q_0\omega_3 + q_1\omega_2 + q_2\omega_1) \quad (2.24)$$

where ω_1, ω_2, ω_3 are angular velocities in body coordinates.

D. EQUATIONS OF MOTION

1. Particle Dynamics

Dynamics is concerned with the relationship between motion and the forces affecting motion. Before discussing the effects of forces on a particle, we need to identify the information necessary to completely describe the motion of a particle. There are important questions that have to be answered to do this. First, where is the particle located? Figure 1.5 shows a particle and a vector **R**. **R** is the position or location of the particle in the **A** frame. The vector runs from the origin of **A** to the particle.

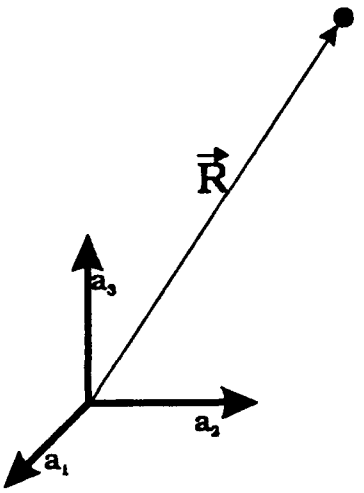


Figure 1.5 Position Vector **R**

The second question that needs to be answered is how fast and in what direction is the particle moving. In other words, what is the change in **R** with respect to time or what is the particle's velocity?

$$ {}^{A}\mathbf{v}^{P} = \frac{{}^{A}d\mathbf{R}}{dt} \tag{2.25} $$

There is one more bit of information needed. That is how is v, the velocity, changing with respect to time. This is the acceleration of the particle.

$$ {}^{A}\mathbf{a}^{P} = \frac{{}^{A}d\mathbf{v}}{dt} = \frac{{}^{A}d^{2}\mathbf{R}}{dt^{2}} \tag{2.26} $$

a. Newton's Laws of Motion

By keeping track of position, velocity, and acceleration, we can accurately describe the motion of the particle. However, this is still kinematics, not dynamics, because there is no mention of how applied forces affect the particle's motion. The foundation for dynamics was laid by Newton with his three laws of motion: [Ref 1: p. 2]

1. A particle at rest remains at rest and a particle in motion remains in uniform, straight-line motion if the applied force is zero.

2. The force on a particle equals the mass of the particle times its inertial acceleration.

3. For every applied force, there is an equal and opposite reaction force.

Newton's second law is commonly written as F = ma, where F is the applied force, m is the particle's mass, and a is the acceleration of the particle. This is the key equation for relating forces to motion. F = ma holds when only a single force is applied, but it also holds when several forces are applied simultaneously. A more general form of the equation is ΣF = ma, where ΣF is the vector sum of the forces. Therefore, given the mass of a particle and all applied forces, the acceleration can be calculated as:

$$^{N}\mathbf{a}^{P} = (\Sigma \mathbf{F})/m \tag{2.27}$$

Now a new velocity can be calculated by integrating the acceleration over time and a new position by integrating the velocity.

$$^{A}\mathbf{v}^{P}(t) = ^{A}\mathbf{v}^{P}(0) + \int_{0}^{t} {}^{A}\mathbf{a}^{P} dt \tag{2.28}$$

$$\mathbf{R}(t) = \mathbf{R}(0) + \int_{0}^{t} {}^{A}\mathbf{v}^{P} dt \tag{2.29}$$

$$\mathbf{R}(t) = \mathbf{R}(0) + {}^{A}\mathbf{v}^{P}(0)t + \int_{0}^{t}\left(\int_{0}^{t} {}^{A}\mathbf{a}^{P} dt\right) dt \tag{2.30}$$

b. Derivatives Between Different Frames

Parameters like velocity can be defined in any frame of reference, but in order for Newton's laws to hold, acceleration must be with respect to an inertial frame of reference. Thus if the velocity is defined in a non-inertial frame, a method is needed to determine the value of the acceleration in an inertial frame. If a vector is defined in the **B** frame, then the operator below can be used to determine the value of the derivative of that vector in the **A** frame. [Ref 1: pp. 5-8]

$$\frac{^{a}d}{dt}(\ ) = \frac{^{b}d}{dt}(\ ) + {}^{a}\boldsymbol{\omega}^{b} \times (\ ) \tag{2.31}$$

2. Rigid Body Dynamics

The previous section described the motion of a particle or point mass. Translational motion was considered, the change in position of the point mass. However, objects in the real world are not point masses, they have shape and size as well as mass. Because of their shape, it is not sufficient to discuss only the displacement of the center of mass. We must also consider changes in the attitude or orientation of the object. That is, we must consider their rotational motion. The relationships in rotational motion are similar in form to their translational counterparts and are summarized in Table 2.2 below. The equation for integration of angular velocity given in this table is incremental because, finite rotations compose as quaternions, not vectors. [Ref 4: p. 267]

Translational	Rotational
mass, m	Inertia, I
Linear momentum, P = mv	Angular momentum, H = Iω
Force, F	Moment, M
Acceleration, a	Angular Acceleration, α
Velocity, v	Angular Velocity, ω
Position, R	Angular Position, θ
$\Sigma F = \frac{dP}{dt} = m\dot{v}$	$\Sigma M = \frac{dH}{dt} = I\dot{\omega} + \dot{I}\omega$
$v = v_o + at$	$\omega = \omega_o + \alpha t$
$R = R_o + v_o t + \frac{1}{2}at^2$	$\Delta\theta = \theta_o + \omega_o t + \frac{1}{2}at^2$

Table 2.2 Corresponding Translation and Rotational Equations

a. Inertia

When dealing with rotational motion, it is necessary to consider the distribution of mass rather than simply the total mass. A body's inertia provides the required information on the distribution of mass about the center of mass. Inertia is expressed in a three by three matrix. The definition for the elements of the inertia matrix is shown below, where body coordinates with origin at the center of mass are assumed for all integrations. [Ref 1: p. 102]

$$I = \begin{bmatrix} \int(y^2+z^2)dm & -\int yx\,dm & -\int zx\,dm \\ -\int xy\,dm & \int(x^2+z^2)dm & -\int zy\,dm \\ -\int xz\,dm & -\int yz\,dm & \int(x^2+y^2)dm \end{bmatrix} \tag{2.32}$$

There are an infinite number of mutually orthogonal axes that can be utilized to determine the value of the matrix above. However, for every body there is at least one set of special axes called *principal* body axes. If this set is used, the non-diagonal elements of the matrix are zero. The inertia matrix in a principal body axis frame is shown below. [Ref 1: pp. 104-106]

$$I = \begin{bmatrix} \int(y^2+z^2)dm & 0 & 0 \\ 0 & \int(x^2+z^2)dm & 0 \\ 0 & 0 & \int(x^2+y^2)dm \end{bmatrix} \tag{2.33}$$

For some geometries there are simple closed form solution to the integrals above. The solutions for a sphere, cylinder, and block are shown on the next page. The short notation for the elements of the inertia matrix is shown below.

$$I = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix} \tag{2.34}$$

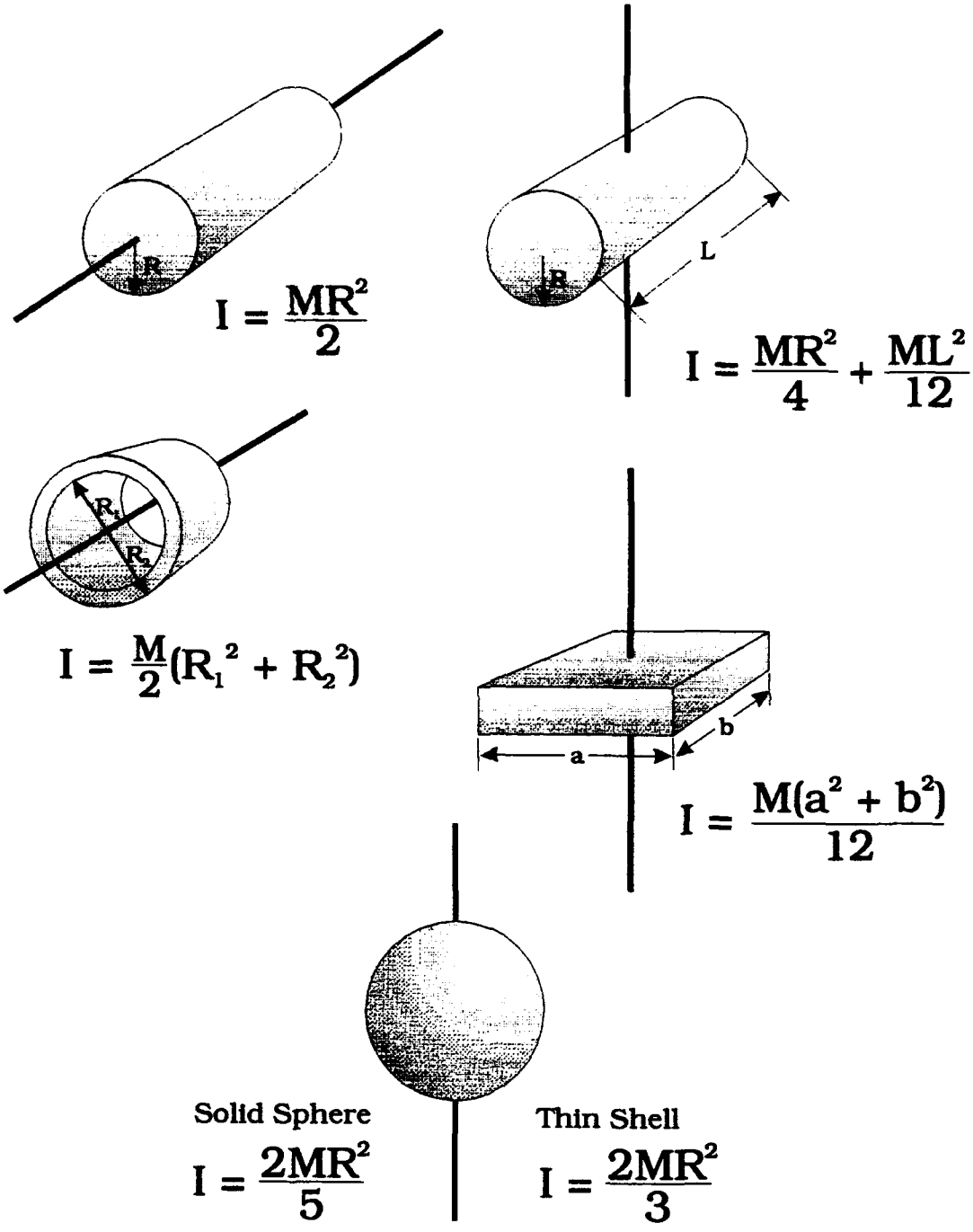


Figure 2.6 Rotational Inertia

b. Euler's Equations of Motion

Euler's equations of motion are a set of three coupled, first-order, nonlinear differential equations that describe the effects of applied moments on the angular rates of a rigid body. Euler's equations are listed below in principal axis coordinates. [Ref 5: p. 112]

$$M_x = I_{xx}\,\dot{\omega}_x + \omega_y\omega_z(I_{zz} - I_{yy}) \tag{2.35}$$

$$M_y = I_{yy}\,\dot{\omega}_y + \omega_x\omega_z(I_{xx} - I_{zz}) \tag{2.36}$$

$$M_z = I_{zz}\,\dot{\omega}_z + \omega_x\omega_y(I_{yy} - I_{xx}) \tag{2.37}$$

If no forces are applied to a body, the linear velocity does not change. It should follow that if no external torques are applied to a rigid body, the angular velocity would not change. Euler's equations show that this is not the case. By solving for the change in angular velocity or angular acceleration and setting external torques equal to zero, this point becomes clear. The equations show that even with no external torques it is possible to change an angular acceleration. There are only three cases when angular acceleration equals zero. The first is trivial, that is when $\omega_x = \omega_y = \omega_z = 0$. The second is when there is inertia symmetry; $I_{xx} = I_{yy} = I_{zz}$. The third case is when all of the angular velocity is about a single principal axis. Other than these three cases, there will be an angular acceleration even without external torques. This effect causes periodic changes in the three angular velocities.

$$\dot{\omega}_x = -\omega_y\omega_z(I_{zz} - I_{yy})/I_{xx} \tag{2.38}$$

$$\dot{\omega}_y = -\omega_x\omega_z(I_{xx} - I_{zz})/I_{yy} \tag{2.39}$$

$$\dot{\omega}_z = -\omega_x\omega_y(I_{yy} - I_{xx})/I_{zz} \tag{2.40}$$

E. SUMMARY

The programs in this thesis are primarily force driven. This chapter covered the key physical concepts that were implemented. These concepts were discussed without regard for computer science. It is imperative to understand these concepts independent of the computer, because physics and computers reside in two different worlds; one continous and the other discrete. There are times when there is not a one to one mapping between

worlds, and things that are straightforward in physics are anything but in the computer's world. To bridge this gap, programmers must understand exactly what should happen, so that he or she can make the necessary adjustments to the computer code.

III. COMPUTER GRAPHICS

A. INTRODUCTION

This chapter discusses the system requirements for the tools developed in this thesis. It also describes the basic procedures required to implement a graphics simulation on a Silicon Graphics computer.

B. SYSTEM REQUIREMENTS

1. Hardware

The programs in this thesis were developed on a Silicon Graphics Elan computer. Below are the specifications for the Elan that the programs were compiled and ran on. These are the minimum system requirements. The graphics board is critical. The programs will not run properly on a system that is not capable of Z Buffering.

1 50 MHz IP20 Processor
FPU: MIPS R4010 Floating Point Processor Chip
CPU: MIPS R4000 Processor Chop
Data Cache: 8 KB
Instruction Cache: 8 KB
Secondary Cache: 1 MB
Main Memory: 64 MB
Iris Audio Processor: revision 10
Graphics Board: GR2-Elan
Mouse

2. Software

Operating System: Silicon Graphics Irix ver 4.05

Compiler: Silicon Graphics C++ Compiler ver 3.0

C. 3D GRAPHICS PROCESS

1. Screen Coordinates

When you are sitting at a computer looking at your monitor, at times it is like you are peering into another world. For all intents and purposes, it is a world complete with its own coordinate system. Figure 3.1 shows the orientation of the coordinate system

utilized by the Silicon Graphics (SGI) computers. Positive Y is up, positive X is to the right, and positive Z is coming out of the screen.

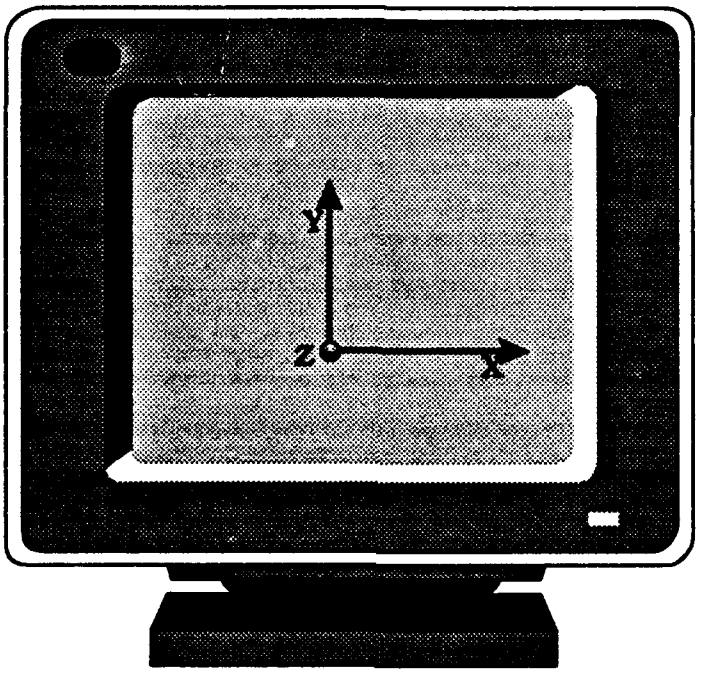


Figure 3.1 Graphics Coordinate System

2. Matrix Stack

The heart of the 3D graphics is the stack of matrices used to construct the scene. The top matrix contains the information necessary to display objects on the screen. SGI's utilize a four by four matrix. The matrix in Figure 3.2 shows the typical matrix structure. The three by three area contains the data for rotations. It is very similar to the rotation matrices generated by an Euler rotation, but there is one significant difference. The matrix is the transpose of the Euler rotation matrix in Chapter II. The reason for this is simple. Matrix multiplication is not commutative. In Chapter 2, a rotation matrix R would multiply a vector V to perform the rotation. The order would be R * V. On SGI's the rotation matrix does not multiply, it premultiplies the matrix on the top of stack S. Its order is S * R instead of R * S. Thus the transpose must be used to have the same effect.

1	0	0	0
0	1	0	0
0	0	1	0
1	4	2	1

Figure 3.2 SGI Matrix Structure

The first three cells of the bottom row in Figure 3.2 show the elements of the matrix that contains the x, y, and z coordinates of object being drawn. The matrix above contains the following values for the object's location: $x = 1$, $y = 4$, $z = 2$. The last column of this matrix is always $(0, 0, 0, 1)^T$ as shown, and is needed to properly combine rotation translation into one *homogenous transform* matrix.

3. Double Buffering

Moving scenes are constructed from a sequence of still pictures or *frames*. In order to get smooth-looking motion, the graphics display system must be able to produce at least 30 frames per second. There are different methods for displaying these frames. SGI's utilize a display technique called *double buffering*. This approach uses two buffers; a *draw buffer* and a *display buffer*. The display buffer contains the data for the frame currently being displayed. The draw buffer is where the images for the next frame to be viewed are stored. When all of the images for the next frame are stored in the draw buffer, the two buffers switch roles. The draw buffer becomes the display buffer and the display buffer becomes the draw buffer.

4. Viewing Process

In order to view a 3D graphics scene on a SGI, there are certain steps that must be followed. This section outlines these steps.

a. Initializing Graphics System

Before any images can be displayed the system must be placed in the graphics mode. This process is called *initializing*. This process determines basic settings for graphics system like location and size of the viewing window. Additionally, the SGI is configured for various desired modes of operation such as double buffering. Finally, input devices like a mouse, keyboard, or spaceball are queued up so the system can receive data from them. For details on the actual configuration utilized refer to Appendix J.

b. Setting Perspective

Perspective is a SGI function that determines how and which objects are to be displayed. Unlike initializing, which occurs once at the beginning of the graphics application, the perspective is set at the beginning of every frame. The function is shown below with the four parameters it requires. [Ref 6: pp. 7-4 - 7-6]

perspective(*Field of View, Aspect Ratio, Near-Clipping-Plane, Far-Clipping-Plane*)

The arguments of this function are defined as follows:

(1) Field of View - This is the width of the scene being viewed in tenths of degrees. A typical value is 450 (45°).

(2) Aspect Ratio - This determines the height (y direction) to width (x direction) ratio or the viewing screen. A value of one means that length takes up as many pixels in the x directions as it does in the y direction. However, 1.25 instead of one is typically used. This is because the pixels on the screen are not square. Their height is 25% longer than their width. Therefore to make objects appear to have the proper dimensions a 1.25 ratio is used.

(3) Near-Clipping-Plane - This determines the location of the near-clipping-plane. Objects that are closer to the observer than the near-clipping-plane are not displayed. These objects are considered in front of the near-clipping-plane. Objects that are farther away from to the observer than the near-clipping-plane are displayed. These objects are considered behind the near-clipping-plane.

(4) Far-Clipping-Plane - This determines the location of the far-clipping-plane. Objects that are closer to the observer than the far-clipping-plane are displayed. These objects are considered in front of the far-clipping-plane. Objects that are farther away from to the observer than the far-clipping-plane are not displayed. These objects are considered behind the far-clipping-plane. Both the near and far clipping planes help to avoid drawing objects that do not affect the scene being constructed.

Before the perspective function can be executed, a unit matrix must be loaded on the stack. A unit matrix has ones down the diagonal and zeros everywhere else.

c. Setting Lookat

Once the perspective is set, the lookat function sets the point of view from which the scene will be observed. The lookat function has the following format. [Ref 6: pp. 7-4 - 7-6]

lookat(*Viewpoint, Reference Point, Twist*)

(1) View point - The view point is the x, y, and z location of the observer

(2) Reference Point - The reference point is the x, y, and z location of the point being observed.

(3) Twist - The twist is amount the scene is rotated in tenths of degrees. Normally it is zero, but there are times when the scene needs to be rotated. Take for example, an observer on an aircraft looking at the horizon. If the aircraft is flying level, no rotation is required. But if the aircraft banks 15° to the right, the scene has the be rotated an equal amount to the left to have the horizon displayed properly.

d. Preparing Objects for Display

The definition of each object is contained in an *off* file. An off file is a text file that contains the color, location, and orientation of all of the polygons that make up the object. There are several functions available for the proper display of objects. The next two functions must be executed after initialization once for each object. The *read_object* function reads the off file into memory and assigns an *OBJECT* pointer to it. OBJECT is a type defined by the SGI graphics library. A special OBJECT pointer is lightobj. It is the object which serves as a lighting source. A light source is required to display the other objects. Once the off file is read into memory, the *ready_object_for_display* function is then executed. An example of a typical execution sequence is shown below.

```
OBJECT* destroyer;
destroyer = read_object("ship.off");
ready_object_for_display(destroyer);
```

e. Drawing Objects

Before objects can be drawn in the draw buffer, the perspective and lookat functions must be executed. At this point, the matrix on top of the matrix stack is set to display the scene. All objects drawn during this frame will use this matrix as a basis. Before proceeding, a copy of the matrix must be made. This is accomplished through the use of a *pushmatrix* function. The pushmatrix function adds a matrix to the top of the matrix, this matrix is identical to the former top.

The top matrix must be modified in order to allow the object to be displayed at the proper location and orientation. The object is moved to the correct location with the *translate* function. The x, y, and z coordinates of the object are the three parameters the translate function requires. The functions premultiplies the matrix on top of the stack by the matrix shown below. [Ref 6: p. C1]

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x & y & z & 1 \end{bmatrix}$$

Figure 3.3 Translation matrix

If the object is not in its original orientation, there is a *rotate* function available to place it in the correct orientation. The amount of the rotation in tenths of degrees and the axis of rotation are the two required arguments. Again the function premultiplies the top of the matrix stack. There are three possible matrices, one for each axis, they are shown below. [Ref 6: p. C2]

$$\text{rotate}(\theta, \text{'X'}) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.1)$$

$$\text{rotate}(\theta, \text{'Y'}) = \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.2)$$

$$\text{rotate}(\theta, \text{'Z'}) = \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.3)$$

Figure 3.4 Rotation matrices

After the translation and rotations, the object is ready to be drawn to the draw buffer. The *display_this_object* function does this. It takes as its only argument the OBJECT pointer for the object to be drawn. This function uses the matrix on top of the stack to draw the object at its proper location and orientation.

The next step is to remove the matrix from the top of the stack. The popmatrix function does this. Finally, the *swapbuffers* function is executed. It switches the draw and display buffers. Below is an example display sequence of commands for a house, tree, car.

```
loadmatrix(unit);    //unit is a unit matrix
perspective(fov, aspect, near, far);
lookat(v_x, v_y, v_z, r_x, r_y, r_z, twist);
pushmatrix();
translate(house_x,house_y,house_z);
diaplay_this_object(house);
popmatrix();
pushmatrix();
translate(tree_x,tree_y,tree_z);
diaplay_this_object(tree);
popmatrix();
pushmatrix();
```

```
translate(car_x,car_y,car_z);
rotate(300,'Y");
diaplay_this_object(car);
popmatrix();
swapbuffers();
```

All of the functions discussed in this section are part of the SGI graphics library. There are more functions available, in the SGI's Graphics Library Programming Guide.

D. SUMMARY

This chapter has discussed graphics fundamentals needed to construct a graphics simulation. An important part of this thesis is that these details are hidden from users; hence the simulation is user friendly. There are only a few graphics calls required. This allows users to concentrate on the physics of the problem instead of graphics.

IV. IMPLEMENTATION

A. INTRODUCTION

Dynamics Programming Library (DPL) is the name given to the C++ Classes implemented by this thesis. This chapter discusses the DPL design considerations. It covers C++ class design and key concepts that were implemented in DPL. Finally, it discusses the method used to validate the software.

B. C++ CLASS DESIGN

1. vector3D Class

Three dimensional vectors and their operations are found throughout dynamics. Position, velocity, acceleration are all usually represented by a three dimensional vector. Since their use is common, a C++ class was developed to support vector operations. The class name is vector3D. The table shows the member variables and their type.

vector3D	
Member Variable	**Type**
x	double
y	double
z	double

Table 4.1 vector3D structure

All of the classes in this thesis use doubles, because floats don't provide sufficient accuracy. When integrating, some of the time steps are very small. If floats are used these small steps may be rounded to zero.

The vector3D class supports many vector operations like dot product, cross product, and normalization. For a complete list of vector3D functions refer to Appendix A.

2. quaternion Class

The quaternion representation for orientation was chosen over the Euler angle representation for two reasons. First, all orientations have a unique solution and there are no singularities in their rates like there are with Euler angles. Secondly, their rate changes are calculated algebraically instead of trigonometrically, which means they execute faster. The member variables for the quaternion class are shown below. For a complete list of quaternion functions refer to Appendix A.

quaternion	
Member Variable	**Type**
q0	double
q1	double
q2	double
q3	double

Table 4.2 quaternion structure

3. matrix3x3 Class

Three-by-three matrices are use often in dynamics. The matrix3x3 was developed to support matrix operations. Operations supported include matrix multiplication, scalar multiplication, and others. For a complete listing of matrix3x3 functions refer to Appendix A. The member variables of the class are listed in the table below.

matrix3x3	
Member Variable	**Type**
m[9]	double

Table 4.3 matrix3x3 structure

Matrices are usually accessed using two values. For example, the cell in the second row, third column of a matrix temp is normally accessed as follows: temp[1][2] (rows and columns start at the number zero instead of one). However, C++ does not support the double bracket convention. Therefore a single value is used instead of two.

To access the cell in the second row, third column of a matrix temp the matrix3x3 format is temp[5].

4. rigid_body Class

To avoid confusion, "rigid_body" (written like this with an underscore) refers to the C++ class and "rigid body" refers to the dynamics interpretation. The rigid_body class is the heart of the DPL. The class is designed to simulate both particle and rigid body dynamics. In addition to the dynamics, the elements necessary for three dimensional graphics support are encapsulated in this class. The table below shows the member variables of the class.

rigid_body	
Member Variable	**Type**
mass	double
location	vector3D*
velocity	vector3D
acceleration	vector3D
force	vector3D
orientation	quaternion
ang_velocity	vector3D
ang_acceleration	vector3D
moment	vector3D
inertia	matrix3x3
size	vector3D
surface_area	double
shape	OBJECT*
display_axis	int
display_shape	int
type_body	int
holder1	vector3D
holder2	vector3D
holder3	quaternion

Table 4.4 rigid_body class structure

a. Physical Attributes

The member variables of this class fall into two main groups. The first group is the physical attributes. These variables are the ones required to accurately describe a rigid body. There are two main coordinate systems used by this class: inertial or world coordinates and body coordinates. The variables are defined in world or body coordinates depending on which one is more appropriate.

(1) mass - This variable contains the mass of the rigid body in kilograms.

(2) location - location is a the position of the rigid body in world coordinates. It is expressed in meters. Location is pointer to a vector3D instead of a vector3D. This is done to allow easy tracking of rigid bodies. If a particular rigid body is the center of interest of a scene, a pointer for the reference point of the scene can be assigned once as the location pointer of the rigid body and it will keep the rigid body in frame without further assignments.

(3) velocity - velocity is defined in world coordinates and is expressed in meters per second.

(4) acceleration - acceleration is defined in world coordinates and is expressed in meters per second squared.

(5) force - force is defined in world coordinates and is expressed in Newtons.

(6) orientation - this variable uses a quaternion to maintain the rigid body's attitude.

(7) ang_velocity - The angular velocity of the rigid body is defined in body coordinates and is in radians per second.

(8) ang_acceleration- The angular acceleration of the rigid body is defined in body coordinates and is in radians per second^2.

(9) moment- The moment applied to the rigid body is defined in body coordinate and is in Newton meters.

(10)inertia - The moments of inertia of this class are expressed in principal axis coordinates.

(11)size - The size is defined as a vector3D to allow individual scaling of length in the x, y, and z direction.

(12)surface_area - The amount of surface is expressed in meters^2.

(13)shape - shape is the 3D graphics representation of the rigid body.

b. Housekeeping Variables

The housekeeping variables are variables that are required for smooth operation of the rigid_body class, but do not have any real physical interpretation.

(1) display_axis - In order to assist in the visual determination of the rigid body's orientation, it is useful to see the body axes. This variable determines whether the body axes are displayed or not. The axes are color coordinated; the x axis is red, the y axis is blue, and the z axis is black.

(2) display_shape - This variable determines whether the rigid body's shape is displayed. Its main use is when scenes are viewed from the point of view of a rigid body. If a scene is viewed from the center of a rigid body and that rigid body is displayed, all that will be observed is the inside of the rigid body. In order to view the scene properly, the rigid body the scene is viewed from should not be displayed.

(3) type_body - This variable allows the system to treat different body types differently. For example, the computation of the elements of the inertia matrix depends on the shape of the body. This variable is used to distinguished between and sphere and a block.

(4) holder variables - holder1, holder2, holder3 are extra slots to store data. The first two are vectors and the third is a quaternion. They are there if more information is required than the physical attributes can provide. For example if both the current and previous values must be maintained, holder1 could be used to keep the previous position. The variables are used by the *update_attached_body* function.

C. GRAPHICS FUNCTIONS

When a scene is viewed, there is only one point of view and one reference point. Because of this, two global variables were defined to manipulate the viewing of the scenes. The *eye* is the variable that represents the view point of the scene. The *target* is the reference point of the scene. Both are vector3D pointers.

The concept of an eye and a target was meant to isolate the user from the low level graphics functions discussed in Chapter III. The user just has to worry about where the eye and target of the scene is, the other details are taken care of by the DPL. The relevant graphics functions are presented in Appendix A.

D. TIME FUNCTIONS

Like the eye and target, time is also treated as a global variable. There are three time variables used: *old_time*, *delta*, *real_time_ratio*. The SGI has a timing signal which the internal clock uses for a time reference. The value of the clock is incremented by one count each system cycle. The value of the clock is maintained in system ticks, not seconds. The old_time variable contains the value of the system clock at the last time check. The delta variable contains the elapsed time in seconds since the last time check. Finally, the real_time_ratio adjusts system time compared to real time. Its value is normally one, which means the system is real time. A real_time_ratio of three would mean that every second of real time corresponds to three seconds of system time (3:1 speedup). If the real_time_ratio is .25, for every four seconds of real time elapsed one second of system time will elapsed (4:1 slowdown).

E. INTEGRATORS

An important goal of this thesis was to develop a system with enough flexibility to handle the different types of problems in dynamics. There are two main approaches to achieving this goal. The first is to develop a collection of very specialized functions to handle the different cases. This approach would produce very accurate functions, because the functions could be tailored to a particular problem, such as, spring motion. However, this would require the user to select from a myriad of functions to chose the correct one. Moreover, a problem arises when none of the functions match a specific case exactly. Namely, which function should be chosen?

The second approach is to make a few general purpose functions to handle any dynamics. This is the approach used in this thesis. The *update_state* functions are the workhorses of the rigid_body class. They take the current *state* of the rigid body and the force applied to determined the next state. The state consists of both translational and

rotational variables like velocity and orientation. The update_state function determines the next state by integrating the equations of motions discussed in Chapter II. The are three different integration techniques used in this thesis.

1. Euler Method

The Euler method is the simplest method used. It is quickest of the three and provides good accuracy in most cases. The only time when it failed to provide good result was with spinning bodies. The Euler equations of motion (equations 2.33 - 35) were unstable using this method. Angular velocities increased without bounds to infinity. The update_state integrator function uses this method. It is a real time integrator.

2. Runga Kutta Fourth Order Method

In order to work with spinning bodies, a different integrator is needed. The Runga Kutta fourth order provides the needed stability. The update_state_rk4 uses this method. As long as the angular velocities don't get very high, this real time method produces good results.

3. Runga Kutta Adaptive Step Method

When more accuracy is needed, the update_state_rk45 can be used. It utilizes a Runga Kutta adaptive step method, also known as a Runga Kutta fourth/fifth order. It is the most stable of the three integrators, but is not real time.

F. SOFTWARE VALIDATION

Once the software is written, the question arises, is it correct? Validation of the software was accomplished at every step in the software's development. The validation for the vector3D, quaternion, and matrix3x3 classes was straightforward. Problems were worked out by hand and then simple programs were written to verify the functions. For example, to validate the cross product function, two arbitrary vectors were chosen and their cross product was calculated by hand. Next a program was written to calculate the cross product using the same vectors. If the program's result was correct, a second test trial was conducted. If it was incorrect, the function was corrected and another trial was performed.

The rigid_body class was not as easy to validate. Simple cases for which the result was well known had to be used to validate the functions. If a variety of simple cases work, it could be assumed that the complicated cases would work also. The first case was a problem in particle dynamics from a physics textbook. What happens to a ball thrown up at 25 m/s neglecting air resistance? The answer is that it rises 32 meters in 2.6 seconds. The system calculated and displayed the same result. A rigid body test was the spin up of a cylinder. The cylinder had a I_{yy} equal to 1600. A moment of 1600 Newton meters was applied for two seconds. The angular velocity increased from 0 to 2.02 rads per sec. The calculated answer for angular velocity was 2.00 rads per sec. Cases using spring motion, centripetal acceleration, and others was tested. All of them produced the expected results.

G. SUMMARY

The issues discussed in this chapter had a significant effect on the development of DPL. In particular the decision to make a few general purpose integrators makes it a much better "what if?" tool. Students can use it to experiment and see what happens when they apply a special set of conditions to a dynamics problem. The graphics implementation sheilds users from the low level graphics details, thus allowing them to focus on the dynamics of the problems.

V. CONCLUSIONS & RECOMMENDATIONS

A. CONCLUSIONS

The primary goal of this thesis was to develop tools that would enhance students understanding of dynamics. A central goal was to keep the applications user friendly. The Dynamics Programming Library was used to write three main applications: euler, gyro, frame. The user's guides for these programs are contained in Appendix B. These programs demonstrate different principles of dynamics. A group of 21 spacecraft engineering students participated in a lab exercise using these programs. Within 20 minutes, the students could run the simulations thus validating their user friendliness. Furthermore, they felt that seeing the dynamics reenforces what they already knew. In particular the "euler" program demonstrated the concept of quaternion rotations, something which all of them had a problem visualizing. The fact that the students believe that it improves their understanding is the final and most important validation of the system.

B. RECOMMENDATIONS

There are three main areas that the DPL can be used for in future research. First, the rigid_body class can be used as a basis for deriving new class of vehicles, like aircraft and wheeled vehicles. These new classes can be used in systems like the Naval Postgraduate School's *NPSNET* to provide vehicles that move realistically. [Ref 7: pp 1 - 18] *Composite_rigid_body* could be another class developed. This class would address the problems of working with multiple rigid bodies. Finnaly the rigid_body class could be used to design graphical control system programs. Such a program could be used to simulate the force that a spacecraft would experence in orbit. The task of the student would be to develop an attitude control system that keeps the vehicle in the correct orientation.

REFERENCES

1. Wiesel, William E., *Spaceflight Dynamics*, McGraw Hill Inc., 1989.

2. Cooke, Joseph M., Zyda, Michael J., Pratt, David R., and McGhee, Robert B., NPSNET: "Flight Simulation Dynamic Modeling Using Quaternions", *Presence*, v. 1, Fall 1992.

3. Kolve, D. I., "Describing an Attitude", paper presented at the 16th Annual AAS Guidnace and Control Conference, Keystone, Colorado, Feb 6 - 10, 1993.

4. Halliday, David, and Resnick, Robert, *Fundamentals of Physics*, John Wiley & Sons, 1988.

5. Agrawal, Brij, N., *Design of Geosynchronous Spacecraft*, Prentice-Hall, 1986

6. McLendon, Patricia, *Graphics Library Programming Guide*, Silicon Graphics Inc., 1991.

7. Zyda, Michael; Pratt, David; Falby, John; Barham, Paul; and Kelleher, Kristen; "1993 NPSNET Research Group Overview", Sep 7, 1993

APPENDIX A

Function Listing

Vector3D Class

Function	Description	Return Value	Example Usage
vector3D()	Default Initializer	vector3D	vector3D velocity;
vector3D(double, double, double)	Initializes a vector3D using three doubles	vector3D	vector3D velocity(12.0, 13.5, 55.9);
vector3D(const vector3D)	Initializes a vector3D using a vector3D	vector3D	vector3D velocity = old_velocity; where old_velocity is a vector3D
operator=(const vector3D&)	Assignment operator for class	vector3D&	velocity = old_velocity; where velocity and old_velocity are vector3D's
operator+(const vector3D&)	Addition operator	vector3D	velocity = velocity + old_velocity; where velocity and old_velocity are vector3D's
operator-(const vector3D&)	Substraction operator	vector3D	velocity = velocity - old_velocity; where velocity and old_velocity are vector3D's
operator*(double)	Scalar mulitplication	vector3D	velocity = velocity * 2; where velocity is a vector3D
operator*(const vector3D&)	Vector dot product	double	dot = velocity * old_velocity; where velocity and old_velocity are vector3D's
operator/(double)	Scalar division	vector3D	velocity = velocity / 2; where velocity is a vector3D
operator^(const vector3D&)	Vector cross product	vector3D	dot = velocity ^ old_velocity; where velocity and old_velocity are vector3D's
operator<<(ostream&, vector3D&)	C++ output	ostream&	cout << velocity; where velocity is a vector3D

operator[](int)	Allows access to individual components of class
	
	Int value range is 0 - 2.
		double	x = velocity[0];
			this returns the first component of the velocity vector and assigns the value to x.
			
			velocity[1] = 33.3;
			this assigns the second component of the velocity vector the value 33.3

magnitude()	Magnitude of vector3D	double	speed = velocity.magnitude(); this assign to the float variable speed a value equal to the magnitude of the velocity vector.
normalize()	Normalized vector3D to unit vector	void	velocity.normalize(); this normalizes the velocity to one.
normalize(double)	Normalized vector3D to a vector equal to the magnitude of the double	void	velocity.normalize(3.0); this normalizes the velocity to three.

Quaternion Class

Function	Description	Return Value	Example Usage
quaternion()	Default Initializer	quaternion	quaternion orientation;
quaternion(double, double, double, double)	Initializes a quaternion using four doubles. The first three doubles are the angles in radians that the axis of rotation makes with the x, y, and z axes respectively. The fourth double is the amount of rotation in radians.	quaternion	quaternion orientation(12.0, 13.5, 55.9, 0.0);
quaternion(const quaternion)	Initializes a quaternion using a quaternion	quaternion	quaternion orientation= old_orientation; where old_orientation is a quaternion

set(double, double, double, double)	Reinitializes a quaternion using four doubles. The first three doubles are the angles in radians that the axis of rotation makes with the x, y, and z axes respectively. The fourth double is the amount of rotation in radians.	void	orientation.set(12.0, 13.5, 55.9, 0.0);
operator=(const quaternion&)	Assignment operator for class	quaternion &	velocity = old_orientation; where velocity and old_orientation are quaternion's
operator+(const quaternion&)	Addition operator	quaternion	orientation= orientation+ old_orientation; where velocity and old_orientation are quaternion's
operator-(const quaternion&)	Substraction operator	quaternion	orientation = orientation - old_orientation ; where orientation and old_orientation are quaternion's
operator*(double)	Scalar mulitplication	quaternion	orientation = orientation * 2; where orientation is a quaternion
operator*(const quaternion&)	Quaternion multiplication	double	dot = orientation * old_orientation ; where orientation and old_orientation are quaternion's
operator<<(ostream&, quaternion&)	C++ output	ostream&	cout << orientation ; where orientation is a quaternion

operator[](int)	Allows access to individual components of class
	
	Int value range is 0 - 3.
		double	x = orientation [0];
			this returns the first component of the orientation quaternion and assigns the value to x.
			
			velocity[1] = 33.3;
			this assigns the second component of the orientation quaternion the value 33.3

magnitude()	Magnitude of quaternion	double	speed = orientation.magnitude(); this assign to the float variable speed a value equal to the magnitude of the orientation quaternion.

normalize()	Normalized quaternion to unit quaternion	void	orientation.normalize(); this normalizes the velocity to one.
rate_of_change(double, double, double)	Using the angular velocity given by 3 doubles to determine the change of rate of the quaternion.	quaternion	rate = position.rate_of_change(1.0, 3.3, 8.0); where rate and position are quaternions and 1.0, 3.3, 8.0 are the x, y, and z components of the angular velocity of position.
update(double, double, double, double)	Calculates the new value of the quaternion using the angular velocity given by first 3 doubles and the time interval in seconds given by the fourth double.	void	position.update(1.0,7.7,0.7,0.01); where position is a quaternion and 1.0, 7.7, 0.7 is the angular velocity and 0.01 is the time interval in seconds
rate_of_change(vector3D)	Using the angular velocity given by the vector3D to determine the change of rate of the quaternion	quaternion	rate = position.rate_of_change(ang_rate); where rate and position are quaternions and ang_rate is a vector3D that contains the angular velocity.
update(vector3D, double)	Calculates the new value of the quaternion using the angular velocity given by vector3D and the time interval in seconds given by the double.	void	position.update(ang_rate,0.01); where position is a quaternion and ang_rate is the angular velocity and 0.01 is the time interval in seconds

Matrix3x3 Class

Function	Description	Return Value	Example Usage
matrix3x3()	Default Initializer	matrix3x3	matrix3x3 rotation;
matrix3x3(double, double, double, double, (double, double, double, double, (double, double, double, double)	Initializes a matrix3x3 using nine doubles	matrix3x3	matrix3x3 rotation(12.0, 13.5, 55.9, 0.0, 12.0, 13.5, 55.9, 0.0, 99.0);
matrix3x3(const matrix3x3)	Initializes a matrix3x3 using a matrix3x3	matrix3x3	matrix3x3 rotation= old_rotation; where old_rotation is a matrix3x3

operator=(const matrix3x3&)	Assignment operator for class	matrix3x3 &	rotation= old_rotation; where rotation and old_rotation are matrix3x3's
operator+(const matrix3x3&)	Addition operator	matrix3x3	velocity = rotation + old_rotation; where rotation and old_rotation are matrix3x3's
operator-(const matrix3x3&)	Substraction operator	matrix3x3	rotation = rotation - old_rotation ; where rotation and old_rotation are matrix3x3's
operator*(double)	Scalar mulitplication	matrix3x3	rotation = rotation * 2; where rotation is a matrix3x3
operator*(const matrix3x3&)	Matrix multiplication	matrix3x3	new_rot = rotation * old_rotation ; where new_rot, rotation and old_rotation are matrix3x3's
operator*(vector3D&)	Matrix multiplication of a vector	vector3D	new_velocity = rotation * old_velocity; where new_velocity and new_velocity are vector3D's and rotation is matrix3x3
operator*(double)	Scalar multiplication	matrix3x3	new_rot = rotation * 12.0 ; where new_rot, rotation are matrix3x3's
DCM_x_rotation(double)	Generates a DCM for a rotation about the x axis	matrix3x3	rotation.DCM_x_rotation(2.0); generates DCM to a rotation of 2 radians about x axis.
DCM_y_rotation(double)	Generates a DCM for a rotation about the x axis	matrix3x3	rotation.DCM_y_rotation(2.0); generates DCM to a rotation of 2 radians about y axis.
DCM_z_rotation(double)	Generates a DCM for a rotation about the x axis	matrix3x3	rotation.DCM_z_rotation(2.0); generates DCM to a rotation of 2 radians about z axis.
DCM_body_to_world(quaternion)	Generates a DCM for converting from body to world coordinates	matrix3x3	rotation.DCM_body_to_world(orientation); generates DCM to rotate from body to world coordinates using the quaternion orientation.

DCM_world_to_body(quaternion)	Generates a DCM for converting from world to body coordinates	matrix3x3	rotation.DCM_world_to_body(orientation); generates DCM to rotate from world to body coordinates using the quaternion orientation.
transpose()	converts the matrix to its transpose	void	inertia.transpose(); where inertia is a matrix3x3
generate_orientation()	Generates a quaternion with an orientation equivilant to the orientation of the matrix	quaternion	new_orientation = DM.generate_orientation(); new_orientation is a quaternion and DM is a matrix3x3.
operator<<(ostream&, matrix3x3&)	C++ output	ostream&	cout << velocity; where velocity is a matrix3x3

operator[](int)	Allows access to individual components of class. Int value range is 0 - 8. 0 is row 1, column 1; 1 is row 1 column 2; etc	double	x = inertia[0];
			this returns the first component of the inertia matrix3x3 and assigns the value to x.
			
			inertia[5] = 33.3;
			this assigns the fifth component of the inertia matrix3x3 the value 33.3

Time Function

Function	Description	Return Value	Example Usage
set_time()	Used to initialize or reset the time	void	set_time();
set_delta()	Determines the elapsed time since the last set_time() or set_delta() command was issued and sets the global variable delta to that value	void	set_delta();
set_delta(double)	Sets delta variable to the value of the double	void	set_delta(.001); sets delta to .001 seconds
read_delta()	Reads the value the delta variable in seconds	double	elasped_time = read_delta();
read_time()	Reads the value the time variable in seconds	double	current_time = read_time();
read_ticks()	Reads the value the time variable in ticks	int	ticks = read_ticks();

reset_time()	Resets the old_time variable to a value equal to the current time minus currrent delta.	void	reset_time();

set_real_time_factor(double)	Set the ratio between real time and the systems internal clock. Valid values are greater than 0. Values less than one slow down operations. Values greater than one speed up operations	void	set_real_time_factor(10);
			This command would make on second of cpu time equal to ten seconds of real time.
			
			set_real_time_factor(0.5);
			This command would make on second of cpu time equal to a half a second of real time.

Graphics Functions

Function	Description	Return Value	Example Usage
initialize()	Initializes the graphics system.	void	initialize();
init_control_window()	Initializes the control window.	void	init_control_window();
main_window()	Returns control to main window.	void	main_window();
control_window()	Returns control to control window.	void	control_window();
clear_control_window()	Clears the control window.	void	clear_control_window();
euler_control_window(int, int, int, int, int, int, int, quaternion, double)	Display controls for euler program. The first three ints are the three angles of rotations. The second three are the axes of rotations. The seventh int is the switch for the "show quaternion" feature. The quaternion is the orientation of the shuttle. The double is the value for theta used by the quaternion rotation.	void	euler_controls(ang1, ang2, ang3, axis1, axis2, axis3, q, orientation, theta);

gyro_control_window(int, int, int, int, vector3D, int, int, int, double, double, double, double)	Display controls for gyro program. The first three ints are the assigned angular velocities. The fourth int determines the shape. The vector3D determines the size. The next three ints are the applied moments. The first double is the duration of applied moment. The second is the magnitude of the moment. The third is the time while the moment is being applied. The fourth is the time of the total session	void	gyro_controls(x, y, z, object, size, m1, m2, m3, duration, mag, elapsed, total);
stat_controls(double, double, double, double, double, double, double, double, vector3D*)	Displays the statistics for the active rigid_body in the gyro program. The first three doubles are the body's angular velocities. The next is the angular momentum. Next are the moments of inertia. Finally the vector3D* is a 300 cycle history of the angular velocity.	void	stat_controls(x, y, z, am, mass, Ix, Iy, Iz, old_av);
frame_control_window(int, int, vector3D, vector3D, vector3D, int)	Display controls for frame program.The parameter in order are viewing level, assignment level, linear velocity, position, angular velocity, viewing axis.	void	frame_control_window(vlevel, flevel, mag, pos, av, vaxis);
view()	Execute the commands for vewing a object to include lookat and swapbuffers	void	view();

view(quaternion, vector3D, int)	Command for viewing a scene from the point of view of a rigid_body. The quaternion is the orientation of the rigid_body. The vector3D is the location of the rigid_body. The int is the axis down which the scene is viewed. The int values are 1, 2, 3 for the positive x, y, and z axes. For views down the negative axes use -1, -2, or -3	void	view(orientation, position, 1); This allows the scene to be viewed from the point of view of a body at a location of position, looking down the positive x axis.
set_eye(double, double, double)	Sets the eye point	void	set_eye(0.0,0.0,0.0);
set_target(double, double, double)	Sets the target point	void	set_target(0.0,0.0,0.0);
attach_eye_to(vector3D*)	Change the address of the global eye vector3D	void	attach_eye_to(near) where near is vector3D*
attach_target_to(vector3D*)	Change the address of the global target vector3D	void	attach_target_to(far) where far is a vector3D*
rotate_view(int)	Rotates the view in tenths of degrees. This function is used in conjunction with the view() function.	void	rotate_view(450); rotates the view 45 degrees.
gravity_status()	Return 0 is gravity is off and 1 if it is on.	int	if(gravity_status()) x++;
set_gravity_on()	Sets gravity variable to 1	void	set_gravity_on();
set_gravity_off()	Sets gravity variable to 0	void	set_gravity_off();
toggle_gravity()	Changes gravity variable to 1 if it is 0 or changes it to 0 if it is 1.	void	toggle_gravity();
air_resistance_status()	Return 0 is air resistance is off and 1 if it is on.	int	if(air_resistance_status()) x++;
set_air_resistance_on()	Sets air resistance variable to 1	void	set_air_resistance_on();
set_air_resistance_off()	Sets air resistance variable to 0	void	set_air_resistance_off();
toggle_air_resistance()	Changes air resistance variable to 1 if it is 0 or changes it to 0 if it is 1.	void	toggle_air_resistance();

Menu Functions

Function	Description	Return Value	Example Usage
initialize_menu()	Initializes the menu system	void	initialize_menu();
queue_test()	Checks the queue to see if a queued device was signalling and returns the code for the device. Code not yet implemented.	int	selection = queue_test();

Rigid_Body Class Functions

Function	Description	Return Value	Example Usage
rigid_body()	Creates a default rigid_body	rigid_body	rigid_body ball;
rigid_body(int)	Creates a rigid_body of one of the following types: 1 - cube, 2 - sphere, 3 - cylender.	rigid_body	rigid_body ball(2);
rigid_body(char*)	Creates a default rigid_body but uses a off file for the shape.	rigid_body ()	rigid_body ball("beach_ball.off");
compute_inertia()	Computes the inertia of the rigid body in principle axis's and assign it to the rigid body object.	void	sphere.compute_inertia(); where sphere is a rigid_body
assign_mass(double)	Assign a mass to the rigid_body	void	ball.assign_mass(10.0); assigns the ball a mass of 10 kg
assign_size(double)	Assign a size to the rigid_body where x, y, and z components are the same	void	block.assign_size(5.0); assign the block x, y, and z equal to 5 meters
assign_size(double, double, double)	Assign a size to the rigid_body	void	block.assign_size(1.0, 2.0, 3.0);
assign_size(vector3D)	Assign a size to the rigid_body	void	assign_size(size) where size is a vector3D
assign_surface_area(double)	Assign a surface area to the rigid_body	void	ball.assign_surface_area(3.5)
assign_inertia(double, double, double)	Assign a inertia to the rigid_body in principle axis's	void	block.assign_inertia(1.0, 2.0, 3.0);

assign_orientation(double, double, double)	Assign orientation to the rigid_body. These are the values for the orientation quaternion	void	block.assign_orientation(1.0, 2.0, 3.0, 8.0);
assign_orientation(quaternion)	Assign orientation to the rigid_body using a quaternion	void	block.assign_orientation(attitude); where attitude is a quaternion
assign_shape(OBJECT*)	Assign a new shape to the rigid_body	void	ball.assign_shape(box); where box is a OBJECT*
assign_location(double, double, double)	Assign location in world coordinates	void	plane.assign_location(0.0, 0.0, 10.0) where plane is a rigid_body
assign_velocity(double, double, double)	Assign velocity in world coordinates	void	plane.assign_velocity(0.0, 0.0, 10.0) where plane is a rigid_body
assign_acceleration(double, double, double)	Assign acceleration in world coordinates	void	plane.assign_acceleration(0.0, 0.0, 10.0) where plane is a rigid_body
assign_ang_velocity(double, double, double)	Assign angular velocity in world coordinates	void	plane.assign_ang_velocity(0.0, 0.0, 10.0) where plane is a rigid_body
assign_ang_acceleration(double, double, double)	Assign angular acceleration in world coordinates	void	plane.assign_ang_acceleration(0.0, 0.0, 10.0) where plane is a rigid_body
assign_force(double, double, double)	Assign force in world coordinates	void	plane.assign_force(0.0, 0.0, 10.0) where plane is a rigid_body
assign_moment(double, double, double)	Assign moment in world coordinates	void	plane.assign_moment(0.0, 0.0, 10.0) where plane is a rigid_body
assign_location(vector3D)	Assign location in world coordinates	void	plane.assign_location(new_value) where plane is a rigid_body and new_value is a vector3D
assign_velocity(vector3D)	Assign velocity in world coordinates	void	plane.assign_velocity(new_value) where plane is a rigid_body and new_value is a vector3D

assign_acceleration(vector3D)	Assign acceleration in world coordinates	void	plane.assign_acceleration(new_value) where plane is a rigid_body and new_value is a vector3D
assign_ang_velocity(vector3D)	Assign angular velocity in world coordinates	void	plane.assign_ang_velocity(new_value) where plane is a rigid_body and new_value is a vector3D
assign_ang_acceleration(vector3D)	Assign angular acceleration in world coordinates	void	plane.assign_ang_acceleration(new_value) where plane is a rigid_body and new_value is a vector3D
assign_force(vector3D)	Assign force in world coordinates	void	plane.assign_force(new_value) where plane is a rigid_body and new_value is a vector3D
assign_moment(vector3D)	Assign moment in world coordinates	void	plane.assign_moment(new_value) where plane is a rigid_body and new_value is a vector3D
assign_velocity_bc(double, double, double)	Assign velocity in body coordinates	void	plane.assign_velocity_bc(0.0, 0.0, 10.0) where plane is a rigid_body
assign_acceleration_bc(double, double, double)	Assign acceleration in body coordinates	void	plane.assign_acceleration_bc(0.0, 0.0, 10.0) where plane is a rigid_body
assign_ang_velocity_bc(double, double, double)	Assign angular velocity in body coordinates	void	plane.assign_ang_velocity_bc(0.0, 0.0, 10.0) where plane is a rigid_body
assign_ang_acceleration_bc(double, double, double)	Assign angular acceleration in body coordinates	void	plane.assign_ang_acceleration_bc(0.0, 0.0, 10.0) where plane is a rigid_body
assign_force_bc(double, double, double)	Assign force in body coordinates	void	plane.assign_force_bc(0.0, 0.0, 10.0) where plane is a rigid_body
assign_moment_bc(double, double, double)	Assign moment in body coordinates	void	plane.assign_moment_bc(0.0, 0.0, 10.0) where plane is a rigid_body

assign_velocity_bc(vector3D)	Assign velocity in body coordinates	void	plane.assign_velocity_bc(new_value) where plane is a rigid_body and new_value is a vector3D
assign_acceleration_bc(vector3D)	Assign acceleration in body coordinates	void	plane.assign_acceleration_bc(new_value) where plane is a rigid_body and new_value is a vector3D
assign_ang_velocity_bc(vector3D)	Assign angular velocity in body coordinates	void	plane.assign_ang_velocity_bc(new_value) where plane is a rigid_body and new_value is a vector3D
assign_ang_acceleration_bc(vector3D)	Assign angular acceleration in body coordinates	void	plane.assign_ang_acceleration_bc(new_value) where plane is a rigid_body and new_value is a vector3D
assign_force_bc(vector3D)	Assign force in body coordinates	void	plane.assign_force_bc(new_value) where plane is a rigid_body and new_value is a vector3D
assign_moment_bc(vector3D)	Assign moment in body coordinates	void	plane.assign_moment_bc(new_value) where plane is a rigid_body and new_value is a vector3D
return_mass()	Returns the mass of the rigid_body	double	weight = ball.return_mass() * 9.81; where ball is a rigid_body and weight is a double
return_location()	Returns location in world coordinates	vector3D	postion = ball.return_location(); where ball is a rigid_body and position is a vector3D
return_location_ptr()	Returns location pointer	vector3D*	position = ball.return_location_ptr(); where ball is a rigid_body and position is a vector3D*
return_velocity()	Returns velocity in world coordinates	vector3D	missle.assign_velocity(plane.return_velocity()); this assign the missile a velocity equal to that of the plane. Where plane and missle are rigid_body's

return_acceleration()	Returns acceleration in world coordinates	vector3D	acc = ball.return_acceleration(); where ball is a rigid_body and acc is a vector3D
return_ang_velocity()	Returns angular velocity in world coordinates	vector3D	av = ball.return_ang_velocity(); where ball is a rigid_body and av is a vector3D
return_ang_acceleration()	Returns angular acceleration in world coordinates	vector3D	aa = ball.return_ang_acceleration(); where ball is a rigid_body and aa is a vector3D
return_force()	Returns force in world coordinates	vector3D	force = ball.return_force(); where ball is a rigid_body and force is a vector3D
return_momen	Returns moment in world coordinates	vector3D	torque = ball.return_moment(); where ball is a rigid_body and torque is a vector3D
return_orientation()	Return orientation	quaternion	attitude = ball.return_orientation(); where ball is a rigid_body and attitude is a quaternion
return_surface_area()	Returns surface area	double	sa = ball.return_surface_area(); where ball is a rigid_body and sa is a double
return_shape()	Returns OBJECT* of rigid_body	OBJECT*	new_shape = ball.return_shape(); where ball is a rigid_body and new_shape is a OBJECT*
return_velocity_bc()	Returns velocity in body coordinates	vector3D	missle.assign_velocity(plane.return_velocity_bc()); this assign the missile a velocity equal to that of the plane. Where plane and missle are rigid_body's
return_acceleration_bc()	Returns acceleration in body coordinates	vector3D	acc = ball.return_acceleration_bc(); where ball is a rigid_body and acc is a vector3D
return_ang_velocity_bc()	Returns angular velocity in body coordinates	vector3D	av = ball.return_ang_velocity_bc(); where ball is a rigid_body and av is a vector3D

return_ang_acceleration_bc()	Returns angular acceleration in body coordinates	vector3D	aa = ball.return_ang_acceleration_bc() ; where ball is a rigid_body and aa is a vector3D
return_force_bc()	Returns force in body coordinates	vector3D	force = ball.return_force_bc(); where ball is a rigid_body and force is a vector3D
return_moment_bc()	Returns moment in body coordinates	vector3D	torque = ball.return_moment_bc(); where ball is a rigid_body and torque is a vector3D
display()	displays the rigid_body	void	ball.display(); where ball is a rigid_body
update_state()	Update the state varibles of the rigid_body using Euler integration	void	ball.update_state(); where ball is a rigid_body.
update_state_rk4()	Update the state varibles of the rigid_body using Runga Kutta Fourth Order.	void	ball.update_state_rk4(); where ball is a rigid_body.
update_state_rk45()	Update the state varibles of the rigid_body using Runga Kutta Fourth/Fifth Order (Adaptive Step). It returns the step size taken in seconds.	double	step = ball.update_state_rk45(); where ball is a rigid_body and step is a double.
attached_body_update_state(rigid_body)	Update the state varibles of the rigid_body using Euler integration. This is used when the motion of the calling rigid_body is defined in the frame of reference of other rigid_body	void	book.update_state(car); where book and car are of the type rigid_body.
zero()	This reinitializes all state variables to their starting conditions	void	shuttle.zero(); where shuttle is a rigid_body
add_axis()	Shows the coordinate axis's of the rigid_body	void	ball.add_axis(); where ball is a rigid_body
remove_axis()	Removes the coordinate axis's of the rigid_body	void	ball.remove_axis(); where ball is a rigid_body

attach_eye()	Attaches the global eye point to the rigid_body.	void	ball.attach_eye(); where ball is a rigid_body
attach_target()	Attaches the global target point to the rigid_body	void	ball.attach_target(); where ball is a rigid_body
set_eye(double, double, double)	Sets the eye point to the given location	void	set_eye(0.0,0.0,0.0);
set_target(double, double, double)	Sets the target point to the given location	void	set_target(99.0,8.4, 98.0);

APPENDIX B

Dynamics Visualizer User Guide

A. EULER

1. Description

This program demonstrates Euler and quaternion rotations. Users can select any Euler sequence desired and the space shuttle will execute the rotation. The angle of each rotation in the sequence is restricted to 5° increments. The program has an option that demonstrates equivalent quaternion rotations. Users select an Euler sequence and a shuttle executes an Euler rotation. For the "equivalent" option, a second shuttle executes a quaternion rotation to achieve the same orientation.

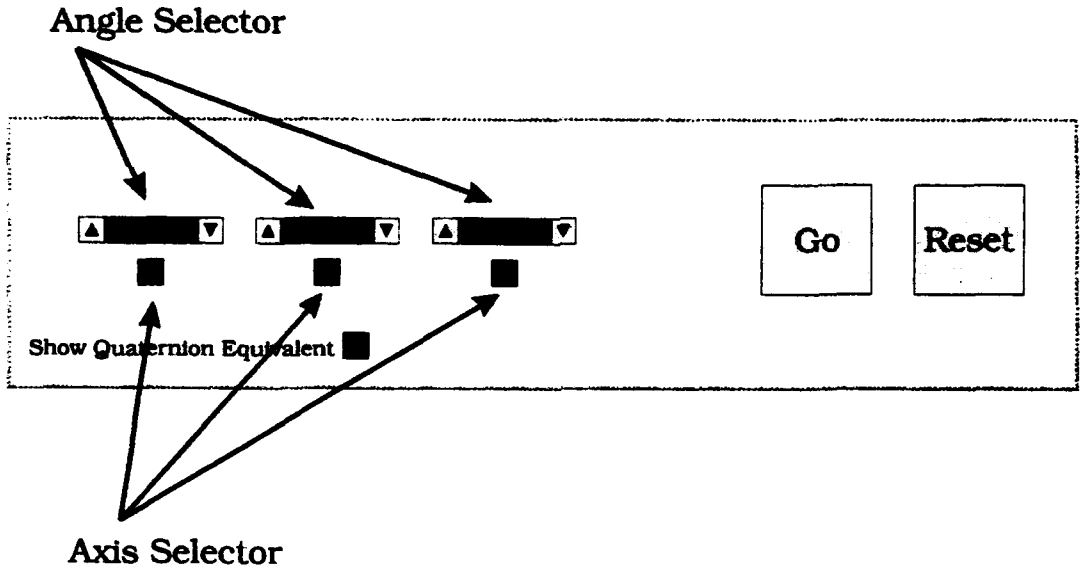


Figure B.1 Euler Control Window

2. Operation

The program is executed by typing **euler** at the prompt. The user selects the three angles for the rotation using the up and down arrows on the angle selectors. Next the axes of rotation are selected. The user can change the axis of rotation by clicking in any of the axis selector boxes. The axes are color coded: Axis 1 is red, axis 2 is blue, axis 3 is

black. The color of the angle and axis selector match the axis of rotation. The colors in Figure B.1 represent a 123 rotation. When the desired sequence has been selected, click on the "GO" button to execute the rotation. Pressing the "Reset" button returns the shuttle to its original orientation.

In order to see the quaternion equivalent rotation, the user selects the "show quaternion equivalent box". Selecting this option causes two shuttles to appear. The one on the left performs Euler rotations and the one on the right performs quaternion rotations. The controls operate the same as before. However, this time when the user executes a sequence, one shuttle performs an Euler rotation and the other performs a quaternion rotation.

3. Suggested Exercise

1. Set the three angles to 10, 30, 60 in order. Execute the following sequences: 123, 321, 121, 313. Notice that each sequence has a different final orientation.
2. Select the "Show Quaternion Equivalent" box and repeat the four rotations.
3. After the rotations are complete, experiment with different sequences and angles.

B. GYRO

1. Description

This program demonstrates basic rigid body dynamics. It allows users to select from three different geometries: a block, sphere, and cylinder. Users can change the size of the rigid body in the x, y, and z directions independently, thus giving them the ability to alter the distribution of mass. The mass itself is set at 1000 kilograms. Once the rigid body is configured, angular velocities can be assigned and moments can be applied. A graph on the bottom of the screen shows the angular velocities of the rigid body in body coordinates. The chart is color coded: X axis is red, Y axis is blue, Z axis is black.

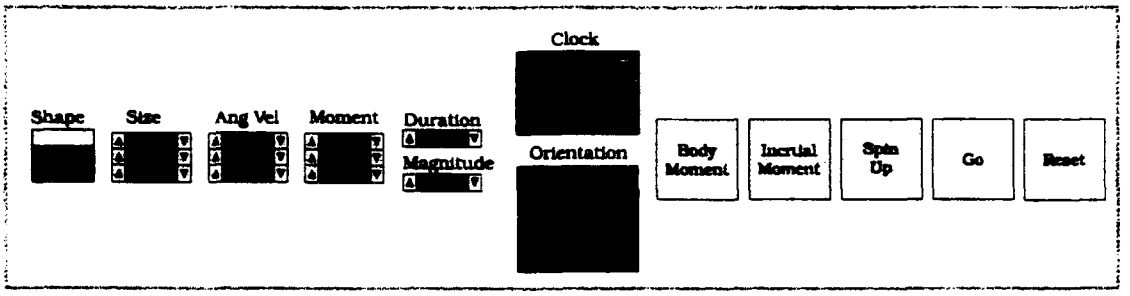


Figure B.2 Gyro Control Window

2. Operation

The program is executed by typing **gyro** at the prompt. Users select the type of rigid body by clicking on one the three choices under the shape field. Size, Ang Vel, Moment, Duration, Magnitude fields use the up and down arrows to change their values. Additionally, Size, Ang Vel, Moment are controlled in the x, y, and z direction independently. Size is the length of the rigid body in meters along its axes. Ang Vel is the assigned angular velocity in radians per sec. To begin the rigid body spinning at this rate, users must press the "Spin Up" button. The Moment field allows users to specify moment in Newton meters. Duration is the length of time that the moment is to be applied in seconds. Magnitude is a scaling factor for the Moment field. To apply the moment, click on either the "Body Moment" or the "Inertial Moment" button, depending on whether the moment is to be applied in body or inertial coordinates.

Gyro is not a real-time system. At times the system slows down significantly. The two clocks display system time. The session clock shows the time since the program was executed. The elapsed clock shows the amount of time the most recent moment was applied. Finally, the "Reset" button resets the rigid body back to its initial condition.

3. Suggested Exercise

1. Select the block set. Set size to x = 2, y = .5, z = 2. Set angular velocity to y = 3 and press spin up. Notice that the blue line appears on the graph on the bottom of the scene.

2. Set angular velocity to y = 3, z = 1 and press spin up. Now there are velocities about all three axes. The y angular velocity is constant due to symmetry. The x and z angular velocities change in a sinusoidal manner.

3. Press reset, set moment to z = 2, duration to 1 and magnitude to 100 (1e+02), then press inertial moment. The block begins to rotate counter clockwise.

4. Press reset, set angular velocity to y = 2, and press spin up. Set moment to z = 2 and press inertial moment. This time motion similar to that in step two is observed.

5. Press reset, set angular velocity to y = 8, and press spin up. Set moment to z = 2 and press inertial moment. This time the moment have much less effect on the motion of the block.

6. Press reset. Change the size to x = .5, y = 2, z = .5 and repeat steps two through five. The moments will have a more profound effect on the motion.

C. FRAME

1. Description

The Frame program shows motion from different frames of reference. The are six different frames: the inertial and one to five. Users are able to define the motion of up of levels one through five. Each *frame level* is defined with respect to the previous frame level. The motion of level one is defined with respect to an inertial frame. The motion of level two is defined with respect to level one, and so on. Once the motion is defined it can be viewed from any frame level.

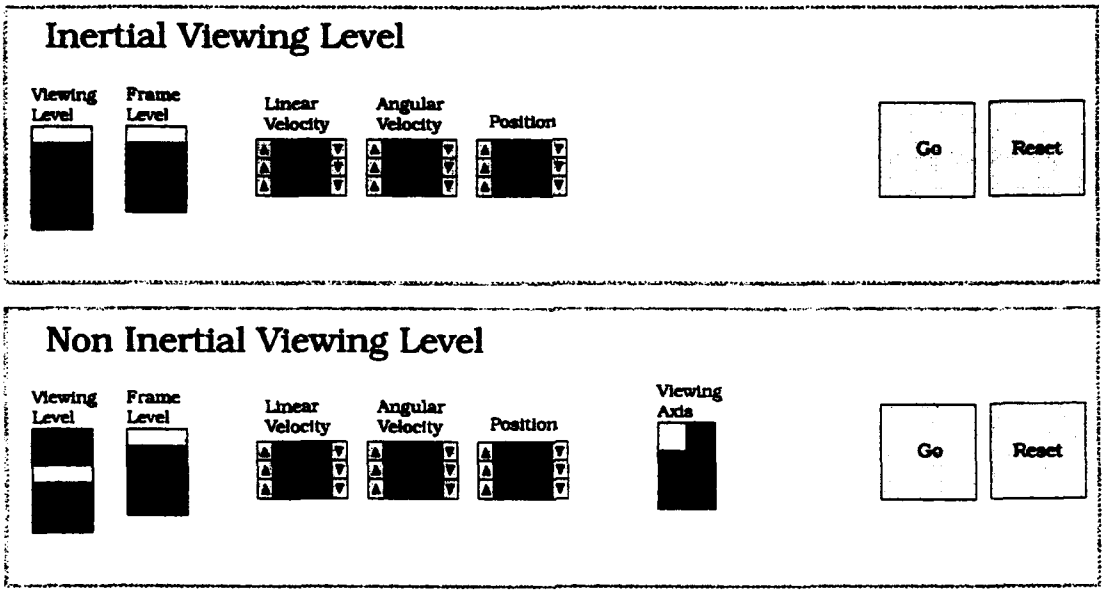


Figure B.3 Frame Control Windows

2. Operation

The program is executed by typing **frame** at the prompt. The scene is initially viewed from the inertial level. The *viewing level* is selects the level from which the scene is viewed. The frame level is the level at which the motion is being defined. A level one through five, three parameters can be specified: linear velocity, position, angular velocity. The top control window in Figure B.3 is the one seen when the viewing level is set to inertial. When any other viewing level is selected, the control window changes to the one on the bottom of the figure. The *viewing axis* is an additional control that allows the user to select the axis down which the scene is viewed. Clicking on the "Go" button starts the scene moving. The "Reset" button returns it to its original configuration.

3. Suggested Exercise

1. Set the follow settings: frame level 1 - angular velocity to y = 1; frame level 2 - position to x = 2, angular velocity to z = .5; frame level 3 - position to y = .5, angular velocity to y = 2. Press go and observe the motion.

2. Observe the motion from the following setting: viewing level - 1, viewing axis - +X; viewing level - 2, viewing axis - -X; viewing level - 2, viewing axis - +Y; viewing level - 3, viewing axis - -Y. Observe the motion for at least twenty seconds at each setting.

APPENDIX C

Graphic Visualizer Code

A. EULER & QUATERNION ROTATIONS

```
#include "base.H"
#include <stdlib.h>

void main()
{
    int section = 0, bypass = 0, NO_GO = 0, go_next = 0;
    vector3D reuse, reuse2, q, lamda, av;
    matrix3x3 rotation;
    quaternion reuse_q;
    double duration = 0.0, xrot = 0.0, yrot = 0.0, zrot = 0.0, rot = 0.0, theta = 0, theta_dot;
    int mx = 0, my = 0, GO = 0, show_q = 0;
    int vaxis[6], axis1 = 0, axis2 = 1, axis3 = 2, ang1 = 0, ang2 = 0, ang3 = 0, show_axis = 0;
    initialize();
    initialize_menu();
    init_control_window();
    main_window();
    rigid_body shuttle(50), shuttle2(50), cylinder(3);
    rigid_body frame(100);
    shuttle.add_axis();
    shuttle2.assign_location(3,0,0);
    set_eye(0.0, 0.0, 8);
    set_target(0.0, 0.0, 0.0);
    set_time();
    while (section != 99)
    {
        section = queue_test();  //return value if keyboard or mouse has input
        set_delta();
        view();
        euler_controls(ang1, ang2, ang3, axis1 + 1, axis2 + 1, axis3 + 1,
            show_q,shuttle.return_orientation(), theta);
        //when using the mouse, the system response is too fast.  bypass allows for a 4 cycle delay before
        //the next mouse input is processed.
        if(bypass > 0 && bypass < 4)
            bypass++;
        else
            bypass = 0;
        if(section > 99999)
        {
            mx = section / 100000;  //decode mouse x coordinate
            my = section - (mx * 100000);  //decode mouse y coordinate
            if (!bypass)
            {
                bypass = 1;
                if(my > 923 && my < 939)
```

```
{
    // adjust rotation angle
    if(mx > 194 && mx < 208)
        ang1 = (ang1 + 5) % 360;
    if(mx > 327 && mx < 342)
        ang2 = (ang2 + 5) % 360;
    if(mx > 460 && mx < 476)
        ang3 = (ang3 + 5) % 360;
    if(mx > 274 && mx < 289)
        ang1 = (ang1 - 5) % 360;
    if(mx > 407 && mx < 421)
        ang2 = (ang2 - 5) % 360;
    if(mx > 540 && mx < 554)
        ang3 = (ang3 - 5) % 360;
}
//select axis for rotation
if(my > 903 && my < 919)
{
    if(mx > 234 && mx < 249)
        axis1 = (axis1 + 1) % 3;
    if(mx > 367 && mx < 382)
        axis2 = (axis2 + 1) % 3;
    if(mx > 500 && mx < 516)
        axis3 = (axis3 + 1) % 3;
    }
    // GO button selected
    if(mx > 740 && mx < 807 && my > 890 && my < 959 && !NO_GO)
    {
        GO = 1;
        NO_GO = 1;
    }

// Show Quaternion Button
if(mx > 261 && mx < 276 && my > 878 && my < 892)
{
    show_q = (show_q + 1) % 2;
    if(show_q)
    {
        shuttle.zero();
        shuttle2.zero();
        shuttle.add_axis();
        shuttle.assign_location(-3,0,0);
        shuttle2.assign_location(3,0,0);
        set_eye(0.0, 0.0, 11);
        set_time();
    }
    else
    {
        shuttle.zero();
        shuttle2.zero();
        shuttle.add_axis();
        set_eye(0.0, 0.0, 8);
    }
    ang1 = 0;
```

```
            ang2 = 0;
            ang3 = 0;
            axis1 = 0;
            axis2 = 1;
            axis3 = 2;
            NO_GO = 0;
            duration = 0.0;
            show_axis = 0;
            GO = 0;
            theta = 0;
        }
        // RESET button selected
        if(mx > 840 && mx < 907 && my > 890 && my < 959)
        {
            if(show_q)
            {
                shuttle.zero();
                shuttle2.zero();
                shuttle.add_axis();
                shuttle.assign_location(-3,0,0);
                shuttle2.assign_location(3,0,0);
                set_eye(0.0, 0.0, 11);
                set_time();
            }
            else
            {
                shuttle.zero();
                shuttle2.zero();
                shuttle.add_axis();
                set_eye(0.0, 0.0, 8);
            }
            //ang1 = 0;
            //ang2 = 0;
            //ang3 = 0;
            axis1 = 0;
            axis2 = 1;
            axis3 = 2;
            NO_GO = 0;
            duration = 0.0;
            show_axis = 0;
            GO = 0;
            theta = 0;
        }
    }
}
main_window();
if(GO == 1)
{   //preparation for first Euler rotation
    reuse = reuse * 0;
    if(ang1)
    reuse[axis1] = .3 * (ang1 / abs(ang1));  //sets angular velocity for rotation
    rot = ang1 * (pi / 180);
    GO++;
    go_next = 0;
```

```
}
if(GO == 2)
{//GO = 2 animates shuttlle for first rotation
    if(go_next)  //stop shuttle for second rotation
    {
        GO++;
        shuttle.assign_ang_velocity_bc(0,0,0);
    }
    else
    {
        if((ang1 > 0 && rot > .3 * read_delta()) || (ang1 < 0 && rot < -.3 * read_delta()))
        {  //rotation not complete
            shuttle.assign_ang_velocity_bc(reuse);
            rot = rot - reuse[axis1] * read_delta();
        }
        else  // last part of  first rotation
        {
            if(ang1)
            reuse = reuse * (rot / (reuse[axis1] * read_delta()));
            shuttle.assign_ang_velocity_bc(reuse);
            go_next = 1;
        }
    }
}
if(GO == 3)
{  //preparation for second rotation
    reuse = reuse * 0;
    if(ang2)
    reuse[axis2] = .3 * (ang2 / abs(ang2));
    rot = ang2 * (pi / 180);
    GO++;
    go_next = 0;
}
if(GO == 4)  // second rotation animation
{
    if(go_next) //stop shuttle and go to third rotation
    {
        GO++;
        shuttle.assign_ang_velocity_bc(0,0,0);
    }
    else
    {
        if((ang2 > 0 && rot > .3 * read_delta()) || (ang2 < 0 && rot < -.3 * read_delta()))
        {// rotate at normal rate
            shuttle.assign_ang_velocity_bc(reuse);
            rot = rot - reuse[axis2] * read_delta();
        }
        else  // last rotation
        {
            if(ang2)
            reuse = reuse * (rot / (reuse[axis2] * read_delta()));
            shuttle.assign_ang_velocity_bc(reuse);
            go_next = 1;
        }
```

```
    }
}
if(GO == 5) //prep for third rotation
{
    reuse = reuse * 0;
    if(ang3)
    reuse[axis3] = .3 * (ang3 / abs(ang3));
    rot = ang3 * (pi / 180);
    GO++;
    go_next = 0;
}
if(GO == 6) //third rotation animation
{
    if(go_next) //third rotations complete
    {
        shuttle.assign_ang_velocity_bc(0,0,0);
        GO++;
        //preparation for quaternion rotation
        theta = 2 * acos((shuttle.return_orientation())[0]); //calculate theta
        theta_dot = theta/5.0;
        q[0] = (shuttle.return_orientation())[1]; //construct q
        q[1] = (shuttle.return_orientation())[2];
        q[2] = (shuttle.return_orientation())[3];
        lamda = q * (1/sin(theta/2)); //calculate lamda, axis of rotation
        lamda.normalize();
        av = lamda * theta_dot; //angular velocity required for quaternion rotation
        shuttle2.assign_ang_velocity(av);
        //calculation for axis of rotation display
        reuse[0] = 0.0;
        reuse[1] = 1.0;
        reuse[2] = 0.0;
        reuse = lamda ^ reuse; //reuse is now the axis of rotation to for the cylinder
        //the cylinder will represent the axis for the shuttle's quaternion rotation
        rot = acos(reuse * lamda); //amout of rotation required for cylinder
        reuse2[0] = 1.0;
        reuse2[1] = 0.0;
        reuse2[2] = 0.0;
        xrot = acos(reuse2 * reuse); //angle made with x axis
        reuse2[0] = 0.0;
        reuse2[1] = 1.0;
        reuse2[2] = 0.0;
        yrot = acos(reuse2 * reuse); //angle made with y axis
        reuse2[0] = 0.0;
        reuse2[1] = 0.0;
        reuse2[2] = 1.0;
        zrot = acos(reuse2 * reuse); //angle made with z axis
        reuse_q.set(xrot,yrot,zrot,rot); //calculate quaternion for orientation of cyilender
        cylinder.zero();
        cylinder.assign_orientation(reuse_q);
        cylinder.assign_size(.05,8,.05);
        cylinder.assign_location(shuttle2.return_location());
        show_axis = 1;
        duration = 5.0; // sets time for quaternion rotation
    }
```

```
            else
            {
                if((ang3 > 0 && rot > .3 * read_delta()) || (ang3 < 0 && rot < -.3 * read_delta()))
                {
                    shuttle.assign_ang_velocity_bc(reuse);
                    rot = rot - reuse[axis3] * read_delta();
                }
                else
                {
                    if(ang3)
                    reuse = reuse * (rot / (reuse[axis3] * read_delta()));
                    shuttle.assign_ang_velocity_bc(reuse);
                    go_next = 1;
                }
            }
        }
        if(GO == 7 && show_q)  //animate quaternion rotation
        {
            if(duration > 0)
            {
                shuttle2.update_state();
                duration = duration - read_delta();
            }
            else
            {  //move shuttle together to eliminate parallax problem
                if((shuttle.return_location())[0] < 0)
                {
                    shuttle.assign_location((shuttle.return_location())[0] + 0.02,0,0);
                    shuttle2.assign_location((shuttle2.return_location())[0] - 0.02,0,0);
                    cylinder.assign_location(shuttle2.return_location());
                }
            }
        }
        if(show_axis && show_q)   cylinder.display();
        shuttle.update_state_rk4();
        shuttle.display();
        if(show_q)  shuttle2.display();
    }
}
```

B. GYROSCOPIC

```
#include "base.H"
#include <stdlib.h>

void main()
{
    int section = 0, bypass = 0, history = 0;
    int NO_GO = 0, go_next = 0, mox = 0, moy = 0, moz = 0;
    vector3D angular_velocity, inertia, size(1,1,1), *old_ang_velocity = new vector3D[300];
    double duration, step = 0.0, am, am1, am2, am3, time = 0.0, mag = 1.0;
    double elapsed_time = 0.0, total_time = 0.0;
    int mx = 0, my = 0, GO = 0, obj = 1, x = 0, y = 0, z = 0;
    initialize();
    initialize_menu();
    init_control_window();
    main_window();
    rigid_body cube(1), ball(2), cylinder(3), shuttle(50);
    rigid_body frame(100), reuse_body;
    reuse_body.assign_shape(ball.return_shape());
    reuse_body.assign_type(2);
    reuse_body.compute_inertia();
    set_target(0.0,0.0,0.0);
    set_eye(0.0, 0.0,10);
    set_time();
    while (section != 99)
    {
        section = queue_test();
        set_delta();
        view();
        //gyro control window data
        gyro_controls(x,y,z,obj,size,mox,moy,moz,time,mag,elapsed_time,total_time);
        if(bypass > 0 && bypass < 4)  //when mouse has been activated this delays next input 4 cycles
            bypass++;
        else
            bypass = 0;
        if(section > 99999)
        {
            mx = section / 100000;  //decode mouse x coordinate
            my = section - (mx * 100000);  //decode mouse y coordinate
            if (!bypass)
            {
                bypass = 1;
                // Size Decrease
                if(mx > 113 && mx < 129)
                {
                    if(my > 936 && my < 953)
                        if(size[0] > 0.15)
                            size[0] = size[0] - .1;
                    if(my > 923 && my < 937)
                        if(size[1] > 0.15)
                            size[1] = size[1] - .1;
                    if(my > 909 && my < 924)
                        if(size[2] > 0.15)
```

```
                size[2] = size[2] - .1;
        reuse_body.assign_size(size);
        reuse_body.compute_inertia();
}
// Size Increase
if(mx > 165 && mx < 182)
{
        if(my > 936 && my < 953)
                size[0] = size[0] + .1;
        if(my > 923 && my < 937)
                size[1] = size[1] + .1;
        if(my > 909 && my < 924)
                size[2] = size[2] + .1;
        reuse_body.assign_size(size);
        reuse_body.compute_inertia();
}
// Angular Velocity Magnitude Decrease
if(mx > 193 && mx < 209)
{
        if(my > 936 && my < 953)
                x--;
        if(my > 923 && my < 937)
                y--;
        if(my > 909 && my < 924)
                z--;
}
// Angular Velocity Magnitude Increase
if(mx > 245 && mx < 262)
{
        if(my > 936 && my < 953)
                x++;
        if(my > 923 && my < 937)
                y++;
        if(my > 909 && my < 924)
                z++;
}
// Moment Magnitude Decrease
if(mx > 381 && mx < 395)
{
        if(my > 936 && my < 953)
                if(time > 0.05) time = time - .1;
        if(my > 909 && my < 924)
                mag = mag / 10.0;
}
// Moment Magnitude Increase
if(mx > 446 && mx < 463)
{
        if(my > 936 && my < 953)
                time = time + .1;
        if(my > 909 && my < 924)
                mag = mag * 10.0;
}

// Duration & Magnitude Decrease
```

```
if(mx > 287 && mx < 301)
{
    if(my > 936 && my < 953)
        mox--;
    if(my > 923 && my < 937)
        moy--;
    if(my > 909 && my < 924)
        moz--;
}
// Duration & Magnitude Increase
if(mx > 339 && mx < 356)
{
    if(my > 936 && my < 953)
        mox++;
    if(my > 923 && my < 937)
        moy++;
    if(my > 909 && my < 924)
        moz++;
}
//Select Shape
if(mx > 19 && mx < 103)
{
    if(my > 936 && my < 953)
    {
        obj = 1;
        reuse_body.assign_shape(ball.return_shape());
        reuse_body.assign_type(2);
        reuse_body.compute_inertia();
    }
    if(my > 923 && my < 937)
    {
        obj = 2;
        reuse_body.assign_shape(cube.return_shape());
        reuse_body.assign_type(1);
        reuse_body.compute_inertia();
    }
    if(my > 909 && my < 924)
    {
        obj = 3;
        reuse_body.assign_shape(cylinder.return_shape());
        reuse_body.assign_type(3);
        reuse_body.compute_inertia();
    }
}

// Body moment button selected
if(mx > 640 && mx < 707 && my > 890 && my < 959 && !NO_GO)
{
    GO = 21;
    duration = time;
    step = 0.0;
    elapsed_time = 0.0;
}
// Inertial Moment button selected
```

```
        if(mx > 740 && mx < 813 && my > 890 && my < 959 && !NO_GO)
        {
            GO = 31;
            duration = time;
            step = 0.0;
            elapsed_time = 0.0;
        }
        // Spin Up button selected
        if(mx > 840 && mx < 907 && my > 890 && my < 959 && !NO_GO)
        {
            GO = 1;
            NO_GO = 1;
        }
        if(GO == 2)
            NO_GO = 0;
        // RESET button selected
        if(mx > 940 && mx < 1007 && my > 890 && my < 959)
        {
            reuse_body.zero();
            reuse_body.assign_size(size);
            x = 0;
            y = 0;
            z = 0;
            mox = 0;
            moy = 0;
            moz = 0;
            NO_GO = 0;
            step = 0.0;
        }
    }
}
main_window();
if(GO == 1)
{//spin up selected
    reuse_body.assign_ang_velocity_bc(x,y,z);
    GO++;
}
if(GO == 21)  //set moment in body coordinates
{
    reuse_body.assign_moment_bc(mox * mag, moy * mag, moz * mag);
    duration = duration - step;
    elapsed_time = elapsed_time + step;
}
if(GO == 31)  //setmoment in world coordinates
{
    reuse_body.assign_moment(mox * mag, moy * mag, moz * mag);
    duration = duration - step;
    elapsed_time = elapsed_time + step;
}
if((GO == 21 || GO == 31) && d··· tion < 0)  //apply moments
{
    duration = 0.0;
    GO = 0;
    NO_GO = 0;
```

```
            }
            frame.display();
            if(duration > 0 &&duration < step) set_delta(duration); //last integration step
            step = reuse_body.update_state_rk45(.000001);
            total_time = total_time + step;
            reuse_body.display();
            angular_velocity = reuse_body.return_ang_velocity_bc();
            inertia = reuse_body.return_inertia();  //calculate angular momentum
            am1 = angular_velocity[0] * inertia[0];
            am2 = angular_velocity[1] * inertia[1];
            am3 = angular_velocity[2] * inertia[2];
            am = sqrt(am1 * am1 + am2 * am2 + am3 * am3); //angular momentum
            stat_controls(angular_velocity[0],angular_velocity[1],angular_velocity[2],
                    am,reuse_body.return_mass(), (reuse_body.return_inertia())[0],
                    (reuse_body.return_inertia())[1], (reuse_body.return_inertia())[2],
                    old_ang_velocity);  //display control window and stats
            //old_ang_velocity is a history of the angular velocities for the last 300 cycles.  Used for graph
            old_ang_velocity[history] = reuse_body.return_ang_velocity_bc();
            history = (history + 1) % 300;

        }
    }
```

C. FRAME OF REFERENCE

```
#include "base.H"
#include <stdlib.h>

void main()
{
    int section = 0, bypass = 0, vlevel = 0, alevel = 0, go_next = 0, i;
    vector3D lvelocity[6], lposition[6], lang_vel[6];
    int mx = 0, my = 0, GO = 0, max_level = 1, vaxis[6];
    initialize();
    initialize_menu();
    init_control_window();
    main_window();
    rigid_body level[6], l1(31), l2(32), l3(33), l4(34), l5(35);  //cubes that represents different levels
    (level[1]).assign_shape(l1.return_shape());
    (level[2]).assign_shape(l2.return_shape());
    (level[3]).assign_shape(l3.return_shape());
    (level[4]).assign_shape(l4.return_shape());
    (level[5]).assign_shape(l5.return_shape());
    set_target(0.0,0.0,0.0);
    set_time();
    bypass = 0;
    GO = 0;
    vlevel = 0;
    alevel = 1;
    max_level = 1;
    for(i=0;i<6;i++)
    {
        (level[i]).zero();
        (level[i]).add_axis();
        (level[i]).assign_size(.2);
        lposition[i] = lposition[i] * 0;  //location of each level
        lvelocity[i] = lvelocity[i] * 0;  //velocity of each level
        lang_vel[i] = lang_vel[i] * 0;  //angular velocity of each level
        vaxis[i] = -1;  // viewing axis for each level
    }
    (level[0]).assign_location(0,0,10);  //level 0 is the inertial frame of reference
    vaxis[0] = -3;
    while (section != 99)
    {
        section = queue_test();
        set_delta();
        frame_controls(vlevel, alevel, lvelocity[alevel], lposition[alevel],
            lang_vel[alevel], vaxis[vlevel]);  //control window
        if(section > 99999 && !bypass)
        {
            bypass = 14;
            mx = section / 100000;  //mouse x coordinate
            my = section - (mx * 100000);  //mouse y coordinate
            //Select Viewing Level
            if(mx > 20 && mx < 96)
            {
                if(my > 922 && my < 940)
```

```
        vlevel = 0;
    if(my > 909 && my < 923)
        vlevel = 1;
    if(my > 896 && my < 910)
        vlevel = 2;
    if(my > 883 && my < 897)
        vlevel = 3;
    if(my > 870 && my < 884)
        vlevel = 4;
    if(my > 857 && my < 871)
        vlevel = 5;
}
//Select level for assignment of attributes
if(mx > 100 && mx < 156)
{
    if(my > 909 && my < 923)
        alevel = 1;
    if(my > 896 && my < 910)
        alevel = 2;
    if(my > 883 && my < 897)
        alevel = 3;
    if(my > 870 && my < 884)
        alevel = 4;
    if(my > 857 && my < 871)
        alevel = 5;
    if(alevel > max_level)
        max_level = alevel;
}

//Decrease linear velocity
if(mx > 261 && mx < 276)
{
    if(my > 909 && my < 923)
        (lvelocity[alevel])[0] = (lvelocity[alevel])[0] - .1;
    if(my > 896 && my < 910)
        (lvelocity[alevel])[1] = (lvelocity[alevel])[1] - .1;
    if(my > 883 && my < 897)
        (lvelocity[alevel])[2] = (lvelocity[alevel])[2] - .1;
    (level[alevel]).assign_velocity(lvelocity[alevel]);
}
//Increase linear velocity
if(mx > 314 && mx < 329)
{
    if(my > 909 && my < 923)
        (lvelocity[alevel])[0] = (lvelocity[alevel])[0] + .1;
    if(my > 896 && my < 910)
        (lvelocity[alevel])[1] = (lvelocity[alevel])[1] + .1;
    if(my > 883 && my < 897)
        (lvelocity[alevel])[2] = (lvelocity[alevel])[2] + .1;
    (level[alevel]).assign_velocity(lvelocity[alevel]);
}
//Decrease magnitude of position
if(mx > 347 && mx < 362)
{
```

```
    if(my > 909 && my < 923)
        (lposition[alevel])[0] = (lposition[alevel])[0] - .1;
    if(my > 896 && my < 910)
        (lposition[alevel])[1] = (lposition[alevel])[1] - .1;
    if(my > 883 && my < 897)
        (lposition[alevel])[2] = (lposition[alevel])[2] - .1;
    //holderl conatains the location in local coordinates
    (level[alevel]).assign_holder1(lposition[alevel]);
    update location in world coordinates
    (level[1]).assign_location((level[1]).return_holder1());
    for(i=2;i<6;i++)
        (level[i]).assign_location((level[i]).return_holder1()+(level[i-1]).return_location());
}

//Increase Magnitude of position
if(mx > 400 && mx < 416)
{
    if(my > 909 && my < 923)
        (lposition[alevel])[0] = (lposition[alevel])[0] + .1;
    if(my > 896 && my < 910)
        (lposition[alevel])[1] = (lposition[alevel])[1] + .1;
    if(my > 883 && my < 897)
        (lposition[alevel])[2] = (lposition[alevel])[2] + .1;
    (level[alevel]).assign_holder1(lposition[alevel]);
    (level[1]).assign_location((level[1]).return_holder1());
    for(i=2;i<6;i++)
        (level[i]).assign_location((level[i]).return_holder1()+(level[i-1]).return_location());
}

//Decrease magnitude of angular velocity
if(mx > 433 && mx < 449)
{
    if(my > 909 && my < 923)
        (lang_vel[alevel])[0] = (lang_vel[alevel])[0] - .1;
    if(my > 896 && my < 910)
        (lang_vel[alevel])[1] = (lang_vel[alevel])[1] - .1;
    if(my > 883 && my < 897)
        (lang_vel[alevel])[2] = (lang_vel[alevel])[2] - .1;
    (level[alevel]).assign_ang_velocity_bc(lang_vel[alevel]);
}
//Increase Magnitude of angular velocity
if(mx > 486 && mx < 502)
{
    if(my > 909 && my < 923)
        (lang_vel[alevel])[0] = (lang_vel[alevel])[0] + .1;
    if(my > 896 && my < 910)
        (lang_vel[alevel])[1] = (lang_vel[alevel])[1] + .1;
    if(my > 883 && my < 897)
        (lang_vel[alevel])[2] = (lang_vel[alevel])[2] + .1;
    (level[alevel]).assign_ang_velocity_bc(lang_vel[alevel]);
}
if(vlevel)
{
    //Select Negative axis for viewing of scene
```

```
                if(mx > 526 && mx < 555)
                {
                    if(my > 909 && my < 923)
                        vaxis[vlevel] = -1;  //-x axis
                    if(my > 896 && my < 910)
                        vaxis[vlevel] = -2;  //-y axis
                    if(my > 883 && my < 897)
                        vaxis[vlevel] = -3;  //-z axis
                }
                //Select Positive axis for viewing of scene
                if(mx > 554 && mx < 583)
                {
                    if(my > 909 && my < 923)
                        vaxis[vlevel] = 1;  //x axis
                    if(my > 896 && my < 910)
                        vaxis[vlevel] = 2;  //y axis
                    if(my > 883 && my < 897)
                        vaxis[vlevel] = 3;  //z axis
                }
            }
            // GO button selected
            if(mx > 740 && mx < 807 && my > 890 && my < 959)
            {
                GO = 1;
            }
            // RESET button selected
            if(mx > 840 && mx < 907 && my > 890 && my < 959)
            {
                GO = 0;
                for(i=1;i<6;i++)
                    (level[i]).assign_holder1(lposition[i]);
                (level[1]).assign_location((level[1]).return_holder1());
                for(i=2;i<6;i++)
                    (level[i]).assign_location((level[i]).return_holder1() +
(level[i-1]).return_location());

            }
        }
        main_window();
        if(GO)
        {
            (level[1]).update_state();
            for(i=2;i<6;i++)
                (level[i]).attached_body_update(level[i-1]);  //integrate frames
        }
        view((level[vlevel]).return_orientation(), (level[vlevel]).return_location(), vaxis[vlevel]);
        for(i=1;i<max_level + 1;i++)
            (level[i]).display(); //view frames
        if (bypass) bypass--;
    }
}
```

APPENDIX D

Rigid_body Code

A. HEADER FILE

```
#ifndef RIGID_BODY_H
#define RIGID_BODY_H
#include <math.h>
#include "vector3D.H"
#include "quaternion.H"
#include "graphics.H"
#include "time.H"
#include "matrix3x3.H"
#include "constants.H"

class rigid_body
{
    double mass;
    vector3D *location;
    vector3D velocity;
    vector3D acceleration;
    vector3D force;
    quaternion orientation;
    vector3D ang_velocity;  // body coordinates
    vector3D ang_acceleration;  // body coordinates
    vector3D moment;  // body coordinates
    matrix3x3 inertia;
    vector3D size;
    double surface_area;
    OBJECT *shape;
    int display_axis;
    int *display_shape;
    int type_body;
    vector3D holder1;
    vector3D holder2;
    quaternion holder3;

public:
    void compute_inertia();
    rigid_body();
    rigid_body(int);
    rigid_body(char*);
    void assign_mass(double);
    void assign_size(double, double, double);
    void assign_size(double);
    void assign_size(vector3D);
    void assign_surface_area(double);
    void assign_inertia(double, double, double);
    void assign_orientation(double, double, double, double);
```

```
void assign_orientation(quaternion);
void assign_shape(OBJECT*);
void assign_type(int);
void assign_holder1(double, double, double);
void assign_holder2(double, double, double);
void assign_holder3(double, double, double, double);
void assign_holder1(vector3D);
void assign_holder2(vector3D);
void assign_holder3(quaternion);

// Assign values to the items using doubles in world coordinates
void assign_location(double, double, double);
void assign_velocity(double, double, double);
void assign_acceleration(double, double, double);
void assign_ang_velocity(double, double, double);
void assign_ang_acceleration(double, double, double);
void assign_force(double, double, double);
void assign_moment(double, double, double);


// Assign values to the items using vector3D in world coordinates
void assign_location(vector3D);
void assign_velocity(vector3D);
void assign_acceleration(vector3D);
void assign_ang_velocity(vector3D);
void assign_ang_acceleration(vector3D);
void assign_force(vector3D);
void assign_moment(vector3D);

// Assign values to the items using doubles in body coordinates
void assign_velocity_bc(double, double, double);
void assign_acceleration_bc(double, double, double);
void assign_ang_velocity_bc(double, double, double);
void assign_ang_acceleration_bc(double, double, double);
void assign_force_bc(double, double, double);
void assign_moment_bc(double, double, double);


// Assign values to the items using vector3D in body coordinates
void assign_velocity_bc(vector3D);
void assign_acceleration_bc(vector3D);
void assign_ang_velocity_bc(vector3D);
void assign_ang_acceleration_bc(vector3D);
void assign_force_bc(vector3D);
void assign_moment_bc(vector3D);


// Return values of the items world coordinates
double return_mass();
vector3D return_inertia();
vector3D return_size();
vector3D return_location();
vector3D* return_location_ptr();
vector3D return_velocity();
```

```
    vector3D return_acceleration();
    quaternion return_orientation();
    vector3D return_ang_velocity();
    vector3D return_ang_acceleration();
    vector3D return_force();
    vector3D return_moment();
    double return_surface_area();
    OBJECT* return_shape();
    int return_type();
    vector3D return_holder1();
    vector3D return_holder2();
    quaternion return_holder3();


    // Return values of the items body coordinates
    vector3D return_velocity_bc();
    vector3D return_acceleration_bc();
    vector3D return_ang_velocity_bc();
    vector3D return_ang_acceleration_bc();
    vector3D return_force_bc();
    vector3D return_moment_bc();


    void display();
    void update_state();
    void update_state_rk4();
    double update_state_rk45(double);
    void add_axis();
    void remove_axis();
    void attach_eye();
    void attach_target();
    void attached_body_update(rigid_body);
    void zero();
    ~rigid_body()
    {
        delete shape;
        delete location;
    }
};

void set_eye(double, double, double);
void set_target(double, double, double);
#endif
```

B. SOURCE FILE

```
#ifndef RIGID_BODY_C
#define RIGID_BODY_C
#include "rigid_body.H"

void rigid_body::compute_inertia()
{
    switch(type_body)
    {
        case 1: //Block
            inertia[0] = mass * ((size[1] * size[1]) + (size[2] * size[2])) / 12.0;
            inertia[4] = mass * ((size[0] * size[0]) + (size[2] * size[2])) / 12.0;
            inertia[8] = mass * ((size[0] * size[0]) + (size[1] * size[1])) / 12.0;
            break;

        case 2: //Shpere
            inertia[0] = mass * .1 * size[1] * size[2];
            inertia[4] = mass * .1 * size[0] * size[2];
            inertia[8] = mass * .1 * size[0] * size[1];
            break;

        case 3: //Cylender
            inertia[0] = mass * ((size[2] * size[2]) / 16.0
                        + (size[1] * size[1]) / 12.0);
            inertia[4] = mass * size[0] * size[2] / 8.0;
            inertia[8] = mass * ((size[0] * size[0]) / 16.0
                        + (size[1] * size[1]) / 12.0);
            break;

    }
}

void rigid_body::assign_type(int t)
{
    type_body = t;
}

//Default Constructor
rigid_body::rigid_body()
{
    location = new vector3D;
    mass = 1000.0;
    size[0] = 1.0;
    size[1] = 1.0;
    size[2] = 1.0;
    surface_area = 1.0;
    shape = NULL;
    type_body = 0;
    display_axis = 0;
    display_shape = new int;
    *display_shape = 1;
}
```

```
rigid_body::rigid_body(char* off_file)   //Constructor using name of off file
{
    location = new vector3D;
    mass = 1000.0;
    size[0] = 1.0;
    size[1] = 1.0;
    size[2] = 1.0;
    surface_area = 1.0;
    shape = read_object(off_file);
    ready_object_for_display(shape);
    type_body = 0;
    display_axis = 0;
    display_shape = new int;
    *display_shape = 1;
}

//Constructor that uses predefined shapes
rigid_body::rigid_body(int n)
{
    location = new vector3D;
    mass = 1000.0;
    size[0] = 1.0;
    size[1] = 1.0;
    size[2] = 1.0;
    surface_area = 1.0;
    switch(n)
    {
        case 100:
        shape = read_object("frame.off");
        type_body = 100;
        break;

        case 1:
        shape = read_object("cube.off");
        mass = 1000.0;
        inertia[0] = 166.7;
        inertia[4] = 166.7;
        inertia[8] = 166.7;
        type_body = 1;
        surface_area = 3.0;
        break;

        case 2:
        shape = read_object("sphere.off");
        type_body = 2;
        mass = 1000.0;
        inertia[0] = 100.0;
        inertia[4] = 100.0;
        inertia[8] = 100.0;
        surface_area = .5;
        break;

        case 3:
        shape = read_object("cylender.off");
```

```
type_body = 3;
mass = 1000.0;
inertia[0] = 145.8;
inertia[4] = 125.0;
inertia[8] = 327.2;
surface_area = 2.0;
break;

case 15:
shape = read_object("f15.off");
mass = 19076;
surface_area = 5.46;
type_body = 15;
break;

case 23:
shape = read_object("rubber_ban.o");
mass = 100;
surface_area = .01;
type_body = 23;
break;

case 31:  //Level 1 cube for frame program
shape = read_object("l1.off");
mass = 100;
surface_area = .01;
type_body = 1;
break;

case 32:  //Level 2 cube for frame program
shape = read_object("l2.off");
mass = 100;
surface_area = .01;
type_body = 1;
break;

case 33:  //Level 3 cube for frame program
shape = read_object("l3.off");
mass = 100;
surface_area = .01;
type_body = 1;
break;

case 34:  //Level 4 cube for frame program
shape = read_object("l4.off");
mass = 100;
surface_area = .01;
type_body = 1;
break;

case 35:  //Level 5 cube for frame program
shape = read_object("l5.off");
mass = 100;
surface_area = .01;
```

```
            type_body = 1;
            break;

            case 50:
            shape = read_object("shuttle.off");
            type_body = 50;
            mass = 1570.8;
            inertia[0] = 327.2;
            inertia[4] = 392.7;
            inertia[8] = 327.2;
            break;

            case 90:
            shape = read_object("ground.off");
            type_body = 90;
            break;

            case 91:
            shape = read_object("floor.off");
            type_body = 91;
            break;

            default:
            shape = NULL;
            type_body = 0;
            break;
        }
        if (type_body)
            ready_object_for_display(shape);
        display_axis = 0;
        display_shape = new int;
        *display_shape = 1;
}

void rigid_body::assign_shape(OBJECT* o)
{
    shape = o;
}

void rigid_body::assign_mass(double n)
{
    mass = n;
}

void rigid_body::assign_surface_area(double n)
{
    surface_area = n;
}

void rigid_body::assign_size(double x, double y, double z)
{
    size[0] = x;
    size[1] = y;
    size[2] = z;
```

```
}

void rigid_body::assign_size(double x)
{
    size[0] = x;
    size[1] = x;
    size[2] = x;
}

void rigid_body::assign_size(vector3D v)
{
    size[0] = v[0];
    size[1] = v[1];
    size[2] = v[2];
}

void rigid_body::assign_location(double x, double y, double z)
{
    (*location)[0] = x;
    (*location)[1] = y;
    (*location)[2] = z;
}

void rigid_body::assign_velocity(double x, double y, double z)
{
    velocity[0] = x;
    velocity[1] = y;
    velocity[2] = z;
}

void rigid_body::assign_acceleration(double x, double y, double z)
{
    acceleration[0] = x;
    acceleration[1] = y;
    acceleration[2] = z;
}

void rigid_body::assign_force(double x, double y, double z)
{
    force[0] = x;
    force[1] = y;
    force[2] = z;
}

void rigid_body::assign_orientation(double x, double y, double z, double w)
{
    orientation[0] = x;
    orientation[1] = y;
    orientation[2] = z;
    orientation[3] = w;
}
```

```
void rigid_body::assign_orientation(quaternion q)
{
    orientation[0] = q[0];
    orientation[1] = q[1];
    orientation[2] = q[2];
    orientation[3] = q[3];
}

void rigid_body::assign_inertia(double x, double y, double z)
{
    inertia[0] =x;
    inertia[4] =y;
    inertia[8] =z;
}

void rigid_body::assign_ang_velocity(double x, double y, double z)
{
    vector3D v(x, y, z);
    matrix3x3 rotation;  //v must be transformed from world to body coordinates
    rotation.DCM_world_to_body(orientation);
    v = rotation * v;
    ang_velocity[0] = v[0];
    ang_velocity[1] = v[1];
    ang_velocity[2] = v[2];
}

void rigid_body::assign_ang_acceleration(double x, double y, double z)
{
    vector3D v(x, y, z);
    matrix3x3 rotation;  //v must be transformed from world to body coordinates
    rotation.DCM_world_to_body(orientation);
    v = rotation * v;
    ang_acceleration[0] = v[0];
    ang_acceleration[1] = v[1];
    ang_acceleration[2] = v[2];
}

void rigid_body::assign_moment(double x, double y, double z)
{
    vector3D v(x, y, z);
    matrix3x3 rotation;  //v must be transformed from world to body coordinates
    rotation.DCM_world_to_body(orientation);
    v = rotation * v;
    moment[0] = v[0];
    moment[1] = v[1];
    moment[2] = v[2];
}

void rigid_body::assign_holder1(double x, double y, double z)
{
    holder1[0] = x;
    holder1[1] = y;
    holder1[2] = z;
}
```

```
void rigid_body::assign_holder2(double x, double y, double z)
{
    holder2[0] = x;
    holder2[1] = y;
    holder2[2] = z;
}

void rigid_body::assign_holder3(double x, double y, double z, double w)
{
    holder3[0] = x;
    holder3[1] = y;
    holder3[2] = z;
    holder3[3] = w;
}

void rigid_body::assign_velocity_bc(double x, double y, double z)
{
    vector3D v(x, y, z);
    matrix3x3 rotation;  //v must be transformed from body to world coordinates
    rotation.DCM_body_to_world(orientation);
    v = rotation * v;
    velocity[0] = v[0];
    velocity[1] = v[1];
    velocity[2] = v[2];
}

void rigid_body::assign_acceleration_bc(double x, double y, double z)
{
    vector3D v(x, y, z);
    matrix3x3 rotation;  //v must be transformed from body to world coordinates
    rotation.DCM_body_to_world(orientation);
    v = rotation * v;
    acceleration[0] = v[0];
    acceleration[1] = v[1];
    acceleration[2] = v[2];
}

void rigid_body::assign_force_bc(double x, double y, double z)
{
    vector3D v(x, y, z);
    matrix3x3 rotation;  //v must be transformed from body to world coordinates
    rotation.DCM_body_to_world(orientation);
    v = rotation * v;
    force[0] = v[0];
    force[1] = v[1];
    force[2] = v[2];
}

void rigid_body::assign_ang_velocity_bc(double x, double y, double z)
{
    ang_velocity[0] = x;
    ang_velocity[1] = y;
    ang_velocity[2] = z;
```

```
}

void rigid_body::assign_ang_acceleration_bc(double x, double y, double z)
{
    ang_acceleration[0] = x;
    ang_acceleration[1] = y;
    ang_acceleration[2] = z;
}

void rigid_body::assign_moment_bc(double x, double y, double z)
{
    moment[0] = x;
    moment[1] = y;
    moment[2] = z;
}

void rigid_body::assign_location(vector3D v)
{
    (*location)[0] = v[0];
    (*location)[1] = v[1];
    (*location)[2] = v[2];
}

void rigid_body::assign_velocity(vector3D v)
{
    velocity[0] = v[0];
    velocity[1] = v[1];
    velocity[2] = v[2];
}

void rigid_body::assign_acceleration(vector3D v)
{
    acceleration[0] = v[0];
    acceleration[1] = v[1];
    acceleration[2] = v[2];
}

void rigid_body::assign_force(vector3D v)
{
    force[0] = v[0];
    force[1] = v[1];
    force[2] = v[2];
}

void rigid_body::assign_ang_velocity(vector3D v)
{
    matrix3x3 rotation;  //v must be transformed from world to body coordinates
    rotation.DCM_world_to_body(orientation);
    v = rotation * v;
    ang_velocity[0] = v[0];
    ang_velocity[1] = v[1];
    ang_velocity[2] = v[2];
}
```

```
void rigid_body::assign_ang_acceleration(vector3D v)
{
    matrix3x3 rotation;  //v must be transformed from world to body coordinates
    rotation.DCM_world_to_body(orientation);
    v = rotation * v;
    ang_acceleration[0] = v[0];
    ang_acceleration[1] = v[1];
    ang_acceleration[2] = v[2];
}


void rigid_body::assign_moment(vector3D v)
{
    matrix3x3 rotation;  //v must be transformed from world to body coordinates
    rotation.DCM_world_to_body(orientation);
    v = rotation * v;
    moment[0] = v[0];
    moment[1] = v[1];
    moment[2] = v[2];
}


void rigid_body::assign_holder1(vector3D v)
{
    holder1 = v;
}

void rigid_body::assign_holder2(vector3D v)
{
    holder2 = v;
}

void rigid_body::assign_holder3(quaternion v)
{
    holder3 = v;
}


void rigid_body::assign_velocity_bc(vector3D v)
{
    matrix3x3 rotation;  //v must be transformed from body to world coordinates
    rotation.DCM_body_to_world(orientation);
    v = rotation * v;
    velocity[0] = v[0];
    velocity[1] = v[1];
    velocity[2] = v[2];
}

void rigid_body::assign_acceleration_bc(vector3D v)
{
    matrix3x3 rotation;  //v must be transformed from body to world coordinates
    rotation.DCM_body_to_world(orientation);
    v = rotation * v;
    acceleration[0] = v[0];
```

```
    acceleration[1] = v[1];
    acceleration[2] = v[2];
}

void rigid_body::assign_force_bc(vector3D v)
{
    matrix3x3 rotation;  //v must be transformed from body to world coordinates
    rotation.DCM_body_to_world(orientation);
    v = rotation * v;
    velocity[0] = v[0];
    velocity[1] = v[1];
    velocity[2] = v[2];
}

void rigid_body::assign_ang_velocity_bc(vector3D v)
{
    ang_velocity[0] = v[0];
    ang_velocity[1] = v[1];
    ang_velocity[2] = v[2];
}

void rigid_body::assign_ang_acceleration_bc(vector3D v)
{
    ang_acceleration[0] = v[0];
    ang_acceleration[1] = v[1];
    ang_acceleration[2] = v[2];
}

void rigid_body::assign_moment_bc(vector3D v)
{
    moment[0] = v[0];
    moment[1] = v[1];
    moment[2] = v[2];
}

int rigid_body::return_type()
{
    return type_body;
}

double rigid_body::return_mass()
{
    return mass;
}

vector3D rigid_body::return_inertia()
{
    vector3D temp;
    temp[0] = inertia[0];
    temp[1] = inertia[4];
    temp[2] = inertia[8];
    return temp;
}
```

```
double rigid_body::return_surface_area()
{
    return surface_area;
}

vector3D rigid_body::return_size()
{
    return size;
}

vector3D rigid_body::return_location()
{
    return *location;
}

vector3D* rigid_body::return_location_ptr()
{
    return location;
}

vector3D rigid_body::return_velocity()
{
    return velocity;
}

vector3D rigid_body::return_acceleration()
{
    return acceleration;
}

vector3D rigid_body::return_force()
{
    return force;
}

quaternion rigid_body::return_orientation()
{
    return orientation;
}

vector3D rigid_body::return_ang_velocity()
{
    vector3D v;
    matrix3x3 rotation;
    rotation.DCM_body_to_world(orientation);
    v = rotation * ang_velocity;
    return v;
}

vector3D rigid_body::return_ang_acceleration()
{
    vector3D v;
    matrix3x3 rotation;
    rotation.DCM_body_to_world(orientation);
```

```
    v = rotation * ang_acceleration;
    return v;
}

vector3D rigid_body::return_moment()
{
    vector3D v;
    matrix3x3 rotation;
    rotation.DCM_body_to_world(orientation);
    v = rotation * moment;
    return v;
}

OBJECT* rigid_body::return_shape()
{
    return shape;
}

vector3D rigid_body::return_holder1()
{
    return holder1;
}

vector3D rigid_body::return_holder2()
{
    return holder2;
}

quaternion rigid_body::return_holder3()
{
    return holder3;
}

vector3D rigid_body::return_velocity_bc()
{
    vector3D v;
    matrix3x3 rotation;
    rotation.DCM_world_to_body(orientation);
    v = rotation * velocity;
    return v;
}

vector3D rigid_body::return_acceleration_bc()
{
    vector3D v;
    matrix3x3 rotation;
    rotation.DCM_world_to_body(orientation);
    v = rotation * acceleration;
    return v;
}

vector3D rigid_body::return_force_bc()
{
    vector3D v;
```

```
    matrix3x3 rotation;
    rotation.DCM_world_to_body(orientation);
    v = rotation * force;
    return v;
}

vector3D rigid_body::return_ang_velocity_bc()
{
    return ang_velocity;
}

vector3D rigid_body::return_ang_acceleration_bc()
{
    return ang_acceleration;
}

vector3D rigid_body::return_moment_bc()
{
    return moment;
}


void rigid_body::update_state()  //Euler method of integration
{
    vector3D gravity(0.0, -9.81, 0.0);
    matrix3x3 rotation;
    double dt = read_delta();
    if(gravity_status())  //check to see if gravity is on, if yes add gravity to force
    {
        force = force + (gravity * mass);
    }
    if(air_resistance_status())  //check to see if air_resistance is on
    {
        double magnitude;
        vector3D direction = velocity * -1.0;
        direction.normalize();
        //formula below is a rough estimate of force due to air resistance
        magnitude = velocity.magnitude() * velocity.magnitude() * 0.12 * surface_area;
        force = force + (direction * magnitude);
    }
    acceleration = force / mass;
    velocity = velocity + (acceleration * dt);
    *location = *location + (velocity * dt) - (acceleration * (0.5 * dt * dt));
    //Eulers equations of motion cannot be used for this method of integration because the are unstable
    //for spinning bodies.  the equations below are used instead.
    ang_acceleration[0] = moment[0] / inertia[0];
    ang_acceleration[1] = moment[1] / inertia[4];
    ang_acceleration[2] = moment[2] / inertia[8];
    ang_velocity = ang_velocity + (ang_acceleration * dt);

    orientation.update(ang_velocity, dt);
    orientation.normalize();
    //all forces and moments are zeroed after each integratior  The user is required to reapply the force
    force[0] = 0.0;
```

```
    force[1] = 0.0;
    force[2] = 0.0;
    moment[0] = 0.0;
    moment[1] = 0.0;
    moment[2] = 0.0;
}

void rigid_body::update_state_rk4() //Runga Kutta forth order integrator
{
//Runga Kutta integrator taken from Numerical Reciepes in C by William Press
double dt = read_delta(); //dt is time step
double hh = dt * .5, h6 = dt / 6;
vector3D ya = ang_velocity, dyma, dyta, yta, dydxa; //required angular velocity variables
vector3D yv = velocity, dymv, dytv, ytv, dydxv; //required velocity variables
vector3D yl = *location, dyml, dytl, ytl, dydxl; //required location variables
quaternion y = orientation, dym, dyt, yt, dydx; //required orientation variables
int i;
vector3D gravity(0.0, -9.81, 0.0);
matrix3x3 rotation;
if(gravity_status()) //check to see if gravity is turned on
{
    force = force + (gravity * mass);
}
if(air_resistance_status()) //check to see if air resistance is turned on

{
    double magnitude;
    vector3D direction = velocity * -1.0;
    direction.normalize();
    magnitude = velocity.magnitude() * velocity.magnitude() * 0.12 * surface_area;
    force = force + (direction * magnitude);
}
acceleration = force / mass;
//calculate initial derivitives
dydxv = acceleration;
dydxl = velocity;
dydxa[0] = (moment[0] - (ang_velocity[1] * ang_velocity[2] * (inertia[8] - inertia[4]))) / inertia[0];
dydxa[1] = (moment[1] - (ang_velocity[0] * ang_velocity[2] * (inertia[0] - inertia[8]))) / inertia[4];
dydxa[2] = (moment[2] - (ang_velocity[1] * ang_velocity[0] * (inertia[4] - inertia[0]))) / inertia[8];
dydx[0] = -0.5*((orientation[1] * ang_velocity[0]) + (orientation[2] * ang_velocity[1]) + (orientation[3] *
ang_velocity[2]));
dydx[1] = 0.5*((orientation[0] * ang_velocity[0]) + (orientation[2] * ang_velocity[2]) - (orientation[3] *
ang_velocity[1]));
dydx[2] = 0.5*((orientation[0] * ang_velocity[1]) + (orientation[3] * ang_velocity[0]) - (orientation[1] *
ang_velocity[2]));
dydx[3] = 0.5*((orientation[0] * ang_velocity[2]) + (orientation[1] * ang_velocity[1]) - (orientation[2] *
ang_velocity[0]));

for(i = 0; i < 3; i++) // compute first midpoint y values
{
    yt[i] = y[i] + hh * dydx[i];
    yta[i] = ya[i] + hh * dydxa[i];
    ytv[i] = yv[i] + hh * dydxv[i];
    ytl[i] = yl[i] + hh * dydxl[i];
```

```
}
yt[3] = y[3] + hh * dydx[3];
//calculate new set of derivitives
dytv = acceleration;
dytl = ytv;
dyt[0] = -0.5*((yt[1] * yta[0]) + (yt[2] * yta[1]) + (yt[3] * yta[2]));
dyt[1] = 0.5*((yt[0] * yta[0]) + (yt[2] * yta[2]) - (yt[3] * yta[1]));
dyt[2] = 0.5*((yt[0] * yta[1]) + (yt[3] * yta[0]) - (yt[1] * yta[2]));
dyt[3] = 0.5*((yt[0] * yta[2]) + (yt[1] * yta[1]) - (yt[2] * yta[0]));
dyta[0] = (moment[0] - (yta[1] * yta[2] * (inertia[8] - inertia[4]))) / inertia[0];
dyta[1] = (moment[1] - (yta[0] * yta[2] * (inertia[0] - inertia[8]))) / inertia[4];
dyta[2] = (moment[2] - (yta[1] * yta[0] * (inertia[4] - inertia[0]))) / inertia[8];

for(i = 0; i < 3; i++)  //calculate second second midpoint y value
{
    yt[i] = y[i] + hh * dyt[i];
    yta[i] = ya[i] + hh * dyta[i];
    ytv[i] = yv[i] + hh * dytv[i];
    ytl[i] = yl[i] + hh * dytl[i];
}
    yt[3] = y[3] + hh * dyt[3];
//calculate new derivitives
dymv = acceleration;
dyml = ytv;
dym[0] = -0.5*((yt[1] * yta[0]) + (yt[2] * yta[1]) + (yt[3] * yta[2]));
dym[1] = 0.5*((yt[0] * yta[0]) + (yt[2] * yta[2]) - (yt[3] * yta[1]));
dym[2] = 0.5*((yt[0] * yta[1]) + (yt[3] * yta[0]) - (yt[1] * yta[2]));
dym[3] = 0.5*((yt[0] * yta[2]) + (yt[1] * yta[1]) - (yt[2] * yta[0]));
dyma[0] = (moment[0] - (yta[1] * yta[2] * (inertia[8] - inertia[4]))) / inertia[0];
dyma[1] = (moment[1] - (yta[0] * yta[2] * (inertia[0] - inertia[8]))) / inertia[4];
dyma[2] = (moment[2] - (yta[1] * yta[0] * (inertia[4] - inertia[0]))) / inertia[8];

for(i = 0; i < 3; i++)  //calculate end point y values
{
    yt[i] = y[i] + dt * dym[i];
    yta[i] = ya[i] + dt * dyma[i];
    ytv[i] = yv[i] + dt * dymv[i];
    ytl[i] = yl[i] + dt * dyml[i];
}
    yt[3] = y[3] + dt * dym[3];


for(i = 0; i < 3; i++)
{
    dym[i] = dym[i] +  dyt[i];
    dyma[i] = dyma[i] +  dyta[i];
    dymv[i] = dymv[i] +  dytv[i];
    dyml[i] = dyml[i] +  dytl[i];
}
dym[3] = dym[3] +  dyt[3];

dytv = acceleration;
dytl = ytv;
dyt[0] = -0.5*((yt[1] * yta[0]) + (yt[2] * yta[1]) + (yt[3] * yta[2]));
```

```
dyt[1] = 0.5*((yt[0] * yta[0]) + (yt[2] * yta[2]) - (yt[3] * yta[1]));
dyt[2] = 0.5*((yt[0] * yta[1]) + (yt[3] * yta[0]) - (yt[1] * yta[2]));
dyt[3] = 0.5*((yt[0] * yta[2]) + (yt[1] * yta[1]) - (yt[2] * yta[0]));
dyta[0] = (moment[0] - (yta[1] * yta[2] * (inertia[8] - inertia[4]))) / inertia[0];
dyta[1] = (moment[1] - (yta[0] * yta[2] * (inertia[0] - inertia[8]))) / inertia[4];
dyta[2] = (moment[2] - (yta[1] * yta[0] * (inertia[4] - inertia[0]))) / inertia[8];


for(i = 0; i < 3; i++)
{
    orientation[i] = y[i] + h6 * (dydx[i] + dyt[i] + 2.0 * dym[i]);
    ang_velocity[i] = ya[i] + h6 * (dydxa[i] + dyta[i] + 2.0 * dyma[i]);
    (*location)[i] = yl[i] + h6 * (dydxl[i] + dytl[i] + 2.0 * dyml[i]);
    velocity[i] = yv[i] + h6 * (dydxv[i] + dytv[i] + 2.0 * dymv[i]);
}
orientation[3] = y[3] + h6 * (dydx[3] + dyt[3] + 2.0 * dym[3]);
orientation.normalize();
force[0] = 0.0;
force[1] = 0.0;
force[2] = 0.0;
moment[0] = 0.0;
moment[1] = 0.0;
moment[2] = 0.0;
}

double rigid_body::update_state_rk45(double eps)  //Runga Kutta Adaptive Step Integrator
{
//Runga Kutta integrator taken from Numerical Reciepes in C by William Press
double PRGOW = -0.20, PSHRNK = -0.25, FCOR = .06666666, dt = read_delta();
double SAFETY = 0.9, ERRCON = 6.0e-4, xsav = dt, htry = dt;
double P = ang_velocity[0], Q = ang_velocity[1], R = ang_velocity[2];
double hh = dt * .5, h6 = dt / 6, h, dtl, hhl, h6l, errmax;
vector3D ya = ang_velocity, dyma, dyta, yta, dydxa, dysava, ysava, ytempa, ytemp2a;
vector3D yv = velocity, dymv, dytv, ytv, dydxv, dysavv, ysavv, ytempv, ytemp2v;
vector3D yl = *location, dyml, dytl, ytl, dydxl, dysavl, ysavl, ytempl, ytemp2l;
quaternion y = orientation, dym, dyt, yt, dydx, dysav, ysav, ytemp, ytemp2;
int i;
vector3D gravity(0.0, -9.81, 0.0);
matrix3x3 rotation;
if(gravity_status())
{
    force = force + (gravity * mass);
}
if(air_resistance_status())
{
    double magnitude;
    vector3D direction = velocity * -1.0;
    direction.normalize();
    magnitude = velocity.magnitude() * velocity.magnitude() * 0.12 * surface_area;
    force = force + (direction * magnitude);
}
acceleration = force / mass;

for(i = 0; i < 3; i++)
```

```
{
    ysav[i] = y[i];
    dysav[i] = dydx[i];
    ysava[i] = ya[i];
    dysava[i] = dydxa[i];
    ysavv[i] = yv[i];
    dysavv[i] = dydxv[i];
    ysavl[i] = yl[i];
    dysavl[i] = dydxl[i];

}
ysav[3] = y[3];
dysav[3] = dydx[3];
h = htry;

for(;;) {
hh = 0.5 * h;
dtl = hh;
hhl = dtl * 0.5;
h6l = dtl / 6.0;
dydxv = acceleration;
dydxl = velocity;
dydxa[0] = (moment[0] - (ang_velocity[1] * ang_velocity[2] * (inertia[8] - inertia[4]))) / inertia[0];
dydxa[1] = (moment[1] - (ang_velocity[0] * ang_velocity[2] * (inertia[0] - inertia[8]))) / inertia[4];
dydxa[2] = (moment[2] - (ang_velocity[1] * ang_velocity[0] * (inertia[4] - inertia[0]))) / inertia[8];
dydx[0] = -0.5*((orientation[1] * ang_velocity[0]) + (orientation[2] * ang_velocity[1]) + (orientation[3] *
ang_velocity[2]));
dydx[1] = 0.5*((orientation[0] * ang_velocity[0]) + (orientation[2] * ang_velocity[2]) - (orientation[3] *
ang_velocity[1]));
dydx[2] = 0.5*((orientation[0] * ang_velocity[1]) + (orientation[3] * ang_velocity[0]) - (orientation[1] *
ang_velocity[2]));
dydx[3] = 0.5*((orientation[0] * ang_velocity[2]) + (orientation[1] * ang_velocity[1]) - (orientation[2] *
ang_velocity[0]));

for(i = 0; i < 3; i++)
{
    yt[i] = ysav[i] + hhl * dydx[i];
    yta[i] = ysava[i] + hhl * dydxa[i];
    ytv[i] = ysavv[i] + hhl * dydxv[i];
    ytl[i] = ysavl[i] + hhl * dydxl[i];
}
yt[3] = ysav[3] + hhl * dydx[3];

dytv = acceleration;
dytl = ytv;
dyt[0] = -0.5*((yt[1] * yta[0]) + (yt[2] * yta[1]) + (yt[3] * yta[2]));
dyt[1] = 0.5*((yt[0] * yta[0]) + (yt[2] * yta[2]) - (yt[3] * yta[1]));
dyt[2] = 0.5*((yt[0] * yta[1]) + (yt[3] * yta[0]) - (yt[1] * yta[2]));
dyt[3] = 0.5*((yt[0] * yta[2]) + (yt[1] * yta[1]) - (yt[2] * yta[0]));
dyta[0] = (moment[0] - (yta[1] * yta[2] * (inertia[8] - inertia[4]))) / inertia[0];
dyta[1] = (moment[1] - (yta[0] * yta[2] * (inertia[0] - inertia[8]))) / inertia[4];
dyta[2] = (moment[2] - (yta[1] * yta[0] * (inertia[4] - inertia[0]))) / inertia[8];

for(i = 0; i < 3; i++)
```

```
{
    yt[i] = ysav[i] + hh1 * dyt[i];
    yta[i] = ysava[i] + hh1 * dyta[i];
    ytv[i] = ysavv[i] + hh1 * dytv[i];
    ytl[i] = ysavl[i] + hh1 * dytl[i];
}
yt[3] = ysav[3] + hh1 * dyt[3];

dymv = acceleration;
dyml = ytv;
dym[0] = -0.5*((yt[1] * yta[0]) + (yt[2] * yta[1]) + (yt[3] * yta[2]));
dym[1] = 0.5*((yt[0] * yta[0]) + (yt[2] * yta[2]) - (yt[3] * yta[1]));
dym[2] = 0.5*((yt[0] * yta[1]) + (yt[3] * yta[0]) - (yt[1] * yta[2]));
dym[3] = 0.5*((yt[0] * yta[2]) + (yt[1] * yta[1]) - (yt[2] * yta[0]));
dyma[0] = (moment[0] - (yta[1] * yta[2] * (inertia[8] - inertia[4]))) / inertia[0];
dyma[1] = (moment[1] - (yta[0] * yta[2] * (inertia[0] - inertia[8]))) / inertia[4];
dyma[2] = (moment[2] - (yta[1] * yta[0] * (inertia[4] - inertia[0]))) / inertia[8];

for(i = 0; i < 3; i++)
{
    yt[i] = ysav[i] + dt1 * dym[i];
    yta[i] = ysava[i] + dt1 * dyma[i];
    ytv[i] = ysavv[i] + dt1 * dymv[i];
    ytl[i] = ysavl[i] + dt1 * dyml[i];
}
    yt[3] = ysav[3] + dt1 * dym[3];


for(i = 0; i < 3; i++)
{
    dym[i] = dym[i] +  dyt[i];
    dyma[i] = dyma[i] +  dyta[i];
    dymv[i] = dymv[i] +  dytv[i];
    dyml[i] = dyml[i] +  dytl[i];
}
dym[3] = dym[3] +  dyt[3];

dytv = acceleration;
dytl = ytv;
dyt[0] = -0.5*((yt[1] * yta[0]) + (yt[2] * yta[1]) + (yt[3] * yta[2]));
dyt[1] = 0.5*((yt[0] * yta[0]) + (yt[2] * yta[2]) - (yt[3] * yta[1]));
dyt[2] = 0.5*((yt[0] * yta[1]) + (yt[3] * yta[0]) - (yt[1] * yta[2]));
dyt[3] = 0.5*((yt[0] * yta[2]) + (yt[1] * yta[1]) - (yt[2] * yta[0]));
dyta[0] = (moment[0] - (yta[1] * yta[2] * (inertia[8] - inertia[4]))) / inertia[0];
dyta[1] = (moment[1] - (yta[0] * yta[2] * (inertia[0] - inertia[8]))) / inertia[4];
dyta[2] = (moment[2] - (yta[1] * yta[0] * (inertia[4] - inertia[0]))) / inertia[8];


for(i = 0; i < 3; i++)
{
    ytemp[i] = ysav[i] + h61 * (dydx[i] + dyt[i] + 2.0 * dym[i]);
    ytempa[i] = ysava[i] + h61 * (dydxa[i] + dyta[i] + 2.0 * dyma[i]);
    ytempl[i] = ysavl[i] + h61 * (dydxl[i] + dytl[i] + 2.0 * dyml[i]);
    ytempv[i] = ysavv[i] + h61 * (dydxv[i] + dytv[i] + 2.0 * dymv[i]);
```

```
}
ytemp[3] = ysav[3] + h61 * (dydx[3] + dyt[3] + 2.0 * dym[3]);


//assign new y's for second rk4
for(i = 0; i < 3; i++)
{
    y[i] = ytemp[i];
    ya[i] = ytempa[i];
    yl[i] = ytempl[i];
    yv[i] = ytempv[i];
}
y[3] = ytemp[3];

//calculate new set of derivatives
dydxv = acceleration;
dydxl = ytempv;
dydx[0] = -0.5*((ytemp[1] * ytempa[0]) + (ytemp[2] * ytempa[1]) + (ytemp[3] * ytempa[2]));
dydx[1] = 0.5*((ytemp[0] * ytempa[0]) + (ytemp[2] * ytempa[2]) - (ytemp[3] * ytempa[1]));
dydx[2] = 0.5*((ytemp[0] * ytempa[1]) + (ytemp[3] * ytempa[0]) - (ytemp[1] * ytempa[2]));
dydx[3] = 0.5*((ytemp[0] * ytempa[2]) + (ytemp[1] * ytempa[1]) - (ytemp[2] * ytempa[0]));
dydxa[0] = (moment[0] - (ytempa[1] * ytempa[2] * (inertia[8] - inertia[4]))) / inertia[0];
dydxa[1] = (moment[1] - (ytempa[0] * ytempa[2] * (inertia[0] - inertia[8]))) / inertia[4];
dydxa[2] = (moment[2] - (ytempa[1] * ytempa[0] * (inertia[4] - iner  a[0]))) / inertia[8];

for(i = 0; i < 3; i++)
{
    yt[i] = y[i] + hh1 * dydx[i];
    yta[i] = ya[i] + hh1 * dydxa[i];
    ytv[i] = yv[i] + hh1 * dydxv[i];
    ytl[i] = yl[i] + hh1 * dydxl[i];
}
yt[3] = y[3] + hh1 * dydx[3];

dytv = acceleration;
dytl = ytv;
dyt[0] = -0.5*((yt[1] * yta[0]) + (yt[2] * yta[1]) + (yt[3] * yta[2]));
dyt[1] = 0.5*((yt[0] * yta[0]) + (yt[2] * yta[2]) - (yt[3] * yta[1]));
dyt[2] = 0.5*((yt[0] * yta[1]) + (yt[3] * yta[0]) - (yt[1] * yta[2]));
dyt[3] = 0.5*((yt[0] * yta[2]) + (yt[1] * yta[1]) - (yt[2] * yta[0]));
dyta[0] = (moment[0] - (yta[1] * yta[2] * (inertia[8] - inertia[4]))) / inertia[0];
dyta[1] = (moment[1] - (yta[0] * yta[2] * (inertia[0] - inertia[8]))) / inertia[4];
dyta[2] = (moment[2] - (yta[1] * yta[0] * (inertia[4] - inertia[0]))) / inertia[8];

for(i = 0; i < 3; i++)
{
    yt[i] = y[i] + hh1 * dyt[i];
    yta[i] = ya[i] + hh1 * dyta[i];
    ytv[i] = yv[i] + hh1 * dytv[i];
    ytl[i] = yl[i] + hh1 * dytl[i];
}
    yt[3] = y[3] + hh1 * dyt[3];

dymv = acceleration;
```

```
dyml = ytv;
dym[0] = -0.5*((yt[1] * yta[0]) + (yt[2] * yta[1]) + (yt[3] * yta[2]));
dym[1] = 0.5*((yt[0] * yta[0]) + (yt[2] * yta[2]) - (yt[3] * yta[1]));
dym[2] = 0.5*((yt[0] * yta[1]) + (yt[3] * yta[0]) - (yt[1] * yta[2]));
dym[3] = 0.5*((yt[0] * yta[2]) + (yt[1] * yta[1]) - (yt[2] * yta[0]));
dyma[0] = (moment[0] - (yta[1] * yta[2] * (inertia[8] - inertia[4]))) / inertia[0];
dyma[1] = (moment[1] - (yta[0] * yta[2] * (inertia[0] - inertia[8]))) / inertia[4];
dyma[2] = (moment[2] - (yta[1] * yta[0] * (inertia[4] - inertia[0]))) / inertia[8];

for(i = 0; i < 3; i++)
{
    yt[i] = y[i] + dt1 * dym[i];
    yta[i] = ya[i] + dt1 * dyma[i];
    ytv[i] = yv[i] + dt1 * dymv[i];
    ytl[i] = yl[i] + dt1 * dyml[i];
}
    yt[3] = y[3] + dt1 * dym[3];


for(i = 0; i < 3; i++)
{
    dym[i] = dym[i] +  dyt[i];
    dyma[i] = dyma[i] +  dyta[i];
    dymv[i] = dymv[i] +  dytv[i];
    dyml[i] = dyml[i] +  dytl[i];
}
dym[3] = dym[3] +  dyt[3];

dytv = acceleration;
dytl = ytv;
dyt[0] = -0.5*((yt[1] * yta[0]) + (yt[2] * yta[1]) + (yt[3] * yta[2]));
dyt[1] = 0.5*((yt[0] * yta[0]) + (yt[2] * yta[2]) - (yt[3] * yta[1]));
dyt[2] = 0.5*((yt[0] * yta[1]) + (yt[3] * yta[0]) - (yt[1] * yta[2]));
dyt[3] = 0.5*((yt[0] * yta[2]) + (yt[1] * yta[1]) - (yt[2] * yta[0]));
dyta[0] = (moment[0] - (yta[1] * yta[2] * (inertia[8] - inertia[4]))) / inertia[0];
dyta[1] = (moment[1] - (yta[0] * yta[2] * (inertia[0] - inertia[8]))) / inertia[4];
dyta[2] = (moment[2] - (yta[1] * yta[0] * (inertia[4] - inertia[0]))) / inertia[8];


for(i = 0; i < 3; i++)
{
    ytemp2[i] = y[i] + h61 * (dydx[i] + dyt[i] + 2.0 * dym[i]);
    ytemp2a[i] = ya[i] + h61 * (dydxa[i] + dyta[i] + 2.0 * dyma[i]);
    ytemp2l[i] = yl[i] + h61 * (dydxl[i] + dytl[i] + 2.0 * dyml[i]);
    ytemp2v[i] = yv[i] + h61 * (dydxv[i] + dytv[i] + 2.0 * dymv[i]);
}
ytemp2[3] = y[3] + h61 * (dydx[3] + dyt[3] + 2.0 * dym[3]);

//thrid rk4 run
dt1 = h;
hh1 = dt1 * 0.5;
h61 = dt1 / 6.0;
dydxv = acceleration;
dydxl = velocity;
```

```
dydxa[0] = (moment[0] - (ang_velocity[1] * ang_velocity[2] * (inertia[8] - inertia[4]))) / inertia[0];
dydxa[1] = (moment[1] - (ang_velocity[0] * ang_velocity[2] * (inertia[0] - inertia[8]))) / inertia[4];
dydxa[2] = (moment[2] - (ang_velocity[1] * ang_velocity[0] * (inertia[4] - inertia[0]))) / inertia[8];
dydx[0] = -0.5*((orientation[1] * ang_velocity[0]) + (orientation[2] * ang_velocity[1]) + (orientation[3] *
ang_velocity[2]));
dydx[1] = 0.5*((orientation[0] * ang_velocity[0]) + (orientation[2] * ang_velocity[2]) - (orientation[3] *
ang_velocity[1]));
dydx[2] = 0.5*((orientation[0] * ang_velocity[1]) + (orientation[3] * ang_velocity[0]) - (orientation[1] *
ang_velocity[2]));
dydx[3] = 0.5*((orientation[0] * ang_velocity[2]) + (orientation[1] * ang_velocity[1]) - (orientation[2] *
ang_velocity[0]));

for(i = 0; i < 3; i++)
{
    yt[i] = ysav[i] + hh1 * dydx[i];
    yta[i] = ysava[i] + hh1 * dydxa[i];
    ytv[i] = ysavv[i] + hh1 * dydxv[i];
    ytl[i] = ysavl[i] + hh1 * dydxl[i];
}
yt[3] = ysav[3] + hh1 * dydx[3];

dytv = acceleration;
dytl = ytv;
dyt[0] = -0.5*((yt[1] * yta[0]) + (yt[2] * yta[1]) + (yt[3] * yta[2]));
dyt[1] = 0.5*((yt[0] * yta[0]) + (yt[2] * yta[2]) - (yt[3] * yta[1]));
dyt[2] = 0.5*((yt[0] * yta[1]) + (yt[3] * yta[0]) - (yt[1] * yta[2]));
dyt[3] = 0.5*((yt[0] * yta[2]) + (yt[1] * yta[1]) - (yt[2] * yta[0]));
dyta[0] = (moment[0] - (yta[1] * yta[2] * (inertia[8] - inertia[4]))) / inertia[0];
dyta[1] = (moment[1] - (yta[0] * yta[2] * (inertia[0] - inertia[8]))) / inertia[4];
dyta[2] = (moment[2] - (yta[1] * yta[0] * (inertia[4] - inertia[0]))) / inertia[8];

for(i = 0; i < 3; i++)
{
    yt[i] = ysav[i] + hh1 * dyt[i];
    yta[i] = ysava[i] + hh1 * dyta[i];
    ytv[i] = ysavv[i] + hh1 * dytv[i];
    ytl[i] = ysavl[i] + hh1 * dytl[i];
}
yt[3] = ysav[3] + hh1 * dyt[3];

dymv = acceleration;
dyml = ytv;
dym[0] = -0.5*((yt[1] * yta[0]) + (yt[2] * yta[1]) + (yt[3] * yta[2]));
dym[1] = 0.5*((yt[0] * yta[0]) + (yt[2] * yta[2]) - (yt[3] * yta[1]));
dym[2] = 0.5*((yt[0] * yta[1]) + (yt[3] * yta[0]) - (yt[1] * yta[2]));
dym[3] = 0.5*((yt[0] * yta[2]) + (yt[1] * yta[1]) - (yt[2] * yta[0]));
dyma[0] = (moment[0] - (yta[1] * yta[2] * (inertia[8] - inertia[4]))) / inertia[0];
dyma[1] = (moment[1] - (yta[0] * yta[2] * (inertia[0] - inertia[8]))) / inertia[4];
dyma[2] = (moment[2] - (yta[1] * yta[0] * (inertia[4] - inertia[0]))) / inertia[8];

for(i = 0; i < 3; i++)
{
    yt[i] = ysav[i] + dt1 * dym[i];
    yta[i] = ysava[i] + dt1 * dyma[i];
```

```
        ytv[i] = ysavv[i] + dt1 * dymv[i];
        ytl[i] = ysavl[i] + dt1 * dyml[i];
    }
        yt[3] = ysav[3] + dt1 * dym[3];


    for(i = 0; i < 3; i++)
    {
        dym[i] = dym[i] +  dyt[i];
        dyma[i] = dyma[i] +  dyta[i];
        dymv[i] = dymv[i] +  dytv[i];
        dyml[i] = dyml[i] +  dytl[i];
    }
    dym[3] = dym[3] +  dyt[3];

    dytv = acceleration;
    dytl = ytv;
    dyt[0] = -0.5*((yt[1] * yta[0]) + (yt[2] * yta[1]) + (yt[3] * yta[2]));
    dyt[1] = 0.5*((yt[0] * yta[0]) + (yt[2] * yta[2]) - (yt[3] * yta[1]));
    dyt[2] = 0.5*((yt[0] * yta[1]) + (yt[3] * yta[0]) - (yt[1] * yta[2]));
    dyt[3] = 0.5*((yt[0] * yta[2]) + (yt[1] * yta[1]) - (yt[2] * yta[0]));
    dyta[0] = (moment[0] - (yta[1] * yta[2] * (inertia[8] - inertia[4]))) / inertia[0];
    dyta[1] = (moment[1] - (yta[0] * yta[2] * (inertia[0] - inertia[8]))) / inertia[4];
    dyta[2] = (moment[2] - (yta[1] * yta[0] * (inertia[4] - inertia[0]))) / inertia[8];


    for(i = 0; i < 3; i++)
    {
        ytemp[i] = ysav[i] + h61 * (dydx[i] + dyt[i] + 2.0 * dym[i]);
        ytempa[i] = ysava[i] + h61 * (dydxa[i] + dyta[i] + 2.0 * dyma[i]);
        ytempl[i] = ysavl[i] + h61 * (dydxl[i] + dytl[i] + 2.0 * dyml[i]);
        ytempv[i] = ysavv[i] + h61 * (dydxv[i] + dytv[i] + 2.0 * dymv[i]);
    }
    ytemp[3] = ysav[3] + h61 * (dydx[3] + dyt[3] + 2.0 * dym[3]);

    //error determination
    errmax = 0.0;
    for(i = 0; i < 3; i++)
    {
        ytemp[i] = ytemp2[i] - ytemp[i];
        if(errmax < fabs(ytemp[i])) errmax = fabs(ytemp[i]);
        ytempa[i] = ytemp2a[i] - ytempa[i];
        if(errmax < fabs(ytempa[i])) errmax = fabs(ytempa[i]);
        ytempl[i] = ytemp2l[i] - ytempl[i];
        if(errmax < fabs(ytempl[i])) errmax = fabs(ytempl[i]);
        ytempv[i] = ytemp2v[i] - ytempv[i];
        if(errmax < fabs(ytempv[i])) errmax = fabs(ytempv[i]);
    }
    ytemp[3] = ytemp2[3] - ytemp[3];
    if(errmax < fabs(ytemp[3])) errmax = fabs(ytemp[3]);
    errmax /= eps;
    if(errmax < 1.0)
        break;
    h = SAFETY * h * exp(PSHRNK*log(errmax));
```

```
}

for(i = 0; i < 3; i++)
{
     orientation[i] = ytemp2[i] + ytemp[i] * FCOR;
     ang_velocity[i] = ytemp2a[i] + ytempa[i] * FCOR;
     (*location)[i] = ytemp2l[i] + ytempl[i] * FCOR;
     velocity[i] = ytemp2v[i] + ytempv[i] * FCOR;
}
orientation[3] = ytemp2[3] + ytemp[3] * FCOR;
orientation.normalize();
force[0] = 0.0;
force[1] = 0.0;
force[2] = 0.0;
moment[0] = 0.0;
moment[1] = 0.0;
moment[2] = 0.0;
return h;
}

void rigid_body::display()
{
Matrix rt = { 1.0, 0.0, 0.0, 0.0,
          0.0, 1.0, 0.0, 0.0,
          0.0, 0.0, 1.0, 0.0,
          0.0, 0.0, 0.0, 1.0};

Matrix scale = { 1.0, 0.0, 0.0, 0.0,
          0.0, 1.0, 0.0, 0.0,
          0.0, 0.0, 1.0, 0.0,
          0.0, 0.0, 0.0, 1.0};

pushmatrix();
//calculate values for the rotation part of matrix
rt[0][0] = ((orientation[0] * orientation[0]) + (orientation[1] *
          orientation[1]) - (orientation[2] * orientation[2]) -
         (orientation[3] * orientation[3]));
rt[1][0] = 2.0 * ((orientation[1] * orientation[2]) - (orientation[0] *
               orientation[3]));
rt[2][0] = 2.0 * ((orientation[0] * orientation[2]) + (orientation[1] *
               orientation[3]));
rt[0][1] = 2.0 * ((orientation[1] * orientation[2]) + (orientation[0] *
               orientation[3]));
rt[1][1] = ((orientation[0] * orientation[0]) - (orientation[1] *
          orientation[1]) + (orientation[2] * orientation[2]) -
         (orientation[3] * orientation[3]));
rt[2][1] = 2.0 * ((orientation[2] * orientation[3]) - (orientation[0] *
               orientation[1]));
rt[0][2] = 2.0 * ((orientation[1] * orientation[3]) - (orientation[0] *
               orientation[2]));
rt[1][2] = 2.0 * ((orientation[2] * orientation[3]) + (orientation[0] *
               orientation[1]));
rt[2][2] = ((orientation[0] * orientation[0]) - (orientation[1] *
          orientation[1]) - (orientation[2] * orientation[2]) +
```

```
        (orientation[3] * orientation[3]));
//load translation part of matrix
rt[3][0] = (*location)[0];
rt[3][1] = (*location)[1];
rt[3][2] = (*location)[2];
multmatrix(rt);  //perform rotataion and translation of rigid_body

scale[0][0] = size[0];
scale[1][1] = size[1];
scale[2][2] = size[2];
multmatrix(scale);  //scale the rigid_body

if(display_axis && *display_shape)  //display rigid_body's body axes
{
    view_axis();
}
if(*display_shape)  //display shape if eye is not attached to this rigid_body
{
    display_this_object(shape);
}
popmatrix();
}

void rigid_body::add_axis()
{
    display_axis = 1;
}

void rigid_body::remove_axis()
{
    display_axis = 0;
}

void rigid_body::attach_eye()
{
    attach_eye_to(location, display_shape);
}

void rigid_body::attach_target()
{
    attach_target_to(location);
}

void set_eye(double x, double y, double z)
{
    set_eye_to(x, y, z);
}

void set_target(double x, double y, double z)
{
    set_target_to(x, y, z);
}

void rigid_body::zero()
```

```
{
    vector3D zero_vector;
    size[0] = 1.0;
    size[1] = 1.0;
    size[2] = 1.0;
    *location = zero_vector;
    velocity = zero_vector;
    acceleration = zero_vector;
    orientation[0] = 1.0;
    orientation[1] = 0.0;
    orientation[2] = 0.0;
    orientation[3] = 0.0;
    ang_velocity = zero_vector;
    ang_acceleration = zero_vector;
    display_axis = 0;
}

void rigid_body::attached_body_update(rigid_body r)
{
    vector3D av;
    matrix3x3 rotation;
    *location = holder1;  //retrieve local position
    update_state();
    holder1 = *location;  //store local position
    rotation.DCM_body_to_world(r.orientation);
    *location = (rotation * (*location)) + *r.location;  //calculate position in world coordinates
    //make adjustment for rotation of frame of reference with respect to the world
    holder2 = (rotation * r.ang_velocity) + r.holder2;
    //holder2 now contains the sum of the angular velocities of all of the previous rigid_bodies
    av = holder2;
    rotation.DCM_world_to_body(orientation);
    av = rotation * av;  //transforms av to the calling rigid_body's body coordinates
    orientation.update(av,read_delta());  //rotates the rigid_body to account for rotation of previous frames
}

#endif
```

APPENDIX E

Vector3D Code

A. HEADER FILE

```
#ifndef VECTOR3D_H
#define VECTOR3D_H
#include <iostream.h>
#include <math.h>

class vector3D
{
    double x;
    double y;
    double z;
public:
    vector3D();
    vector3D(double, double, double);
    vector3D(const vector3D&);
    vector3D& operator=(const vector3D&);
    vector3D operator+(const vector3D&);
    vector3D operator-(const vector3D&);
    double operator*(const vector3D&);  //dot product
    vector3D operator*(double);  //scalar multiplication
    vector3D operator/(double);  //scalar division
    vector3D operator^(const vector3D&);  //cross product
    double magnitude();
    void normalize();
    void normalize(double);
    friend  ostream& operator<<(ostream&, vector3D&);
    double& operator[](int);
    ~vector3D() { }
};
#endif
```

B. SOURCE FILE

```
#ifndef VECTOR3D_C
#define VECTOR3D_C
#include "vector3D.H"

//default constructor
vector3D::vector3D()
{
    x = 0.0;
    y = 0.0;
    z = 0.0;
}

//constructor using three doubles
vector3D::vector3D(double a, double b, double c)
{
    x = a;
    y = b;
    z = c;
}

//constructor using another vector3D
vector3D::vector3D(const vector3D& v)
{
    x = v.x;
    y = v.y;
    z = v.z;
}

//Assignment operator - the function must return a reference to a vector
//instead of a vector for assignment to work properly
vector3D& vector3D::operator=(const vector3D& v)
{
    x = v.x;
    y = v.y;
    z = v.z;
    return *this;
}

//vector addition operator
vector3D vector3D::operator+(const vector3D& v)
{
    vector3D sum;
    sum.x = v.x + x;
    sum.y = v.y + y;
    sum.z = v.z + z;
    return sum;
}

//vector substraction
vector3D vector3D::operator-(const vector3D& v)
{
    vector3D diff;
```

```
    diff.x = x - v.x;
    diff.y = y - v.y;
    diff.z = z - v.z;
    return diff;
}

//vector dot product
double vector3D::operator*(const vector3D& v)
{
    double dot;
    dot = (v.x * x) + (v.y * y) + (v.z * z);
    return dot;
}

//scalar multiplication
vector3D vector3D::operator*(double n)
{
    vector3D mult;
    mult.x = x * n;
    mult.y = y * n;
    mult.z = z * n;
    return mult;
}

//scalar division - it is the user responsibility to make sure that n is not zero
vector3D vector3D::operator/(double n)
{
    vector3D result;
    result.x = x / n;
    result.y = y / n;
    result.z = z / n;
    return result;
}

//vector cross product
vector3D vector3D::operator^(const vector3D& v)
{
    vector3D cross;
    cross.x = (y * v.z) - (v.y * z);
    cross.y = -((x * v.z) - (v.x * z));
    cross.z = (x * v.y) - (v.x * y);
    return cross;
}

//the << operator is to be used with cout
ostream& operator<<(ostream& os, vector3D& v)
{
    os << (double) v.x << ", " << (double) v.y << ", " << (double) v.z << "\n";
    return os;
}

//allows access to the components of the vector3D.  it must return a reference
//in order for assignment to work
double& vector3D::operator[](int n)
```

```
{
    if (n == 0)
    {
        return x;
    }
    if (n == 1)
    {
        return y;
    }
    if (n == 2)
    {
        return z;
    }

}

//returns the magnitude of the vector
double vector3D::magnitude()
{
    return sqrt((x * x) + (y * y) + (z * z));
}

//normalizes the vector to one
void vector3D::normalize()
{
    double m = sqrt((x * x) + (y * y) + (z * z));
    if (m)
    {
    x = x / m;
    y = y / m;
    z = z / m;
    }
}

//normalize the vector to d
void vector3D::normalize(double d)
{
    double m = sqrt((x * x) + (y * y) + (z * z));
    if (m)
    {
    x = d * x / m;
    y = d * y / m;
    z = d * z / m;
    }
}


#endif
```

APPENDIX F

Quaternion Code

A. HEADER FILE

```
#ifndef QUATERNION_H
#define QUATERNION_H
#include <iostream.h>
#include <math.h>
#include "vector3D.H"


class quaternion
{
    double q0;
    double q1;
    double q2;
    double q3;
public:
    quaternion();
    quaternion(double, double, double, double);
    quaternion(const quaternion&);
    void set(double, double, double, double);
    quaternion& operator=(const quaternion&);
    quaternion operator+(const quaternion&);
    quaternion operator-(const quaternion&);
    quaternion operator*(const quaternion&);
    quaternion operator*(double);
    double& operator[](int);
    double magnitude();
    void normalize();
    quaternion rate_of_change(double, double, double);
    void update(double, double, double, double);
    quaternion rate_of_change(vector3D);
    void update(vector3D, double);
    friend ostream& operator<<(ostream&, quaternion&);
    ~quaternion() { }
};

#endif
```

B. SOURCE FILE

```
#ifndef QUATERNION_C
#define QUATERNION_C
#include "quaternion.H"

//Default Constructor
quaternion::quaternion()
{
    q0 = 1.0;
    q1 = 0.0;
    q2 = 0.0;
    q3 = 0.0;
}

quaternion::quaternion(double angle_x, double angle_y, double angle_z, double rotation)
{
//angle_x, angle_y, angle_z are the angles the axis of rotation make with the coordinate axes in radians
//rotation is the amount of the rotation in radians
    q0 = cosf(0.5*rotation);
    q1 = cosf(angle_x)*sinf(0.5*rotation);
    q2 = cosf(angle_y)*sinf(0.5*rotation);
    q3 = cosf(angle_z)*sinf(0.5*rotation);

}

quaternion::quaternion(const quaternion& q)
{
    q0 = q.q0;
    q1 = q.q1;
    q2 = q.q2;
    q3 = q.q3;
}

void quaternion::set(double angle_x, double angle_y, double angle_z, double rotation)
{
//angle_x, angle_y, angle_z are the angles the axis of rotation make with the coordinate axes in radians
//rotation is the amount of the rotation in radians
    q0 = cosf(0.5*rotation);
    q1 = cosf(angle_x)*sinf(0.5*rotation);
    q2 = cosf(angle_y)*sinf(0.5*rotation);
    q3 = cosf(angle_z)*sinf(0.5*rotation);

}

quaternion& quaternion::operator=(const quaternion& q)
{
    q0 = q.q0;
    q1 = q.q1;
    q2 = q.q2;
    q3 = q.q3;
    return *this;
}
```

```
quaternion quaternion::operator+(const quaternion& q)
{
    quaternion sum;
    sum.q0 = q0 + q.q0;
    sum.q1 = q1 + q.q1;
    sum.q2 = q2 + q.q2;
    sum.q3 = q3 + q.q3;
    return sum;
}

quaternion quaternion::operator-(const quaternion& q)
{
    quaternion diff;
    diff.q0 = q0 - q.q0;
    diff.q1 = q1 - q.q1;
    diff.q2 = q2 - q.q2;
    diff.q3 = q3 - q.q3;
    return diff;
}

quaternion quaternion::operator*(const quaternion& q)
{
    quaternion prod;
    prod.q0 = (q0 * q.q0) - (q1 * q.q1) - (q2 * q.q2) - (q3 * q.q3);
    prod.q1 = (q1 * q.q0) + (q0 * q.q1) - (q3 * q.q2) + (q2 * q.q3);
    prod.q2 = (q2 * q.q0) + (q3 * q.q1) - (q0 * q.q2) + (q1 * q.q3);
    prod.q3 = (q3 * q.q0) + (q2 * q.q1) - (q1 * q.q2) + (q0 * q.q3);
    return prod;
}

quaternion quaternion::operator*(double n)
{
    quaternion prod;
    prod.q0 = q0 * n;
    prod.q1 = q1 * n;
    prod.q2 = q2 * n;
    prod.q3 = q3 * n;
    return prod;
}

double quaternion::magnitude()
{
    return sqrt((q0 * q0) + (q1 * q1) + (q2 * q2) + (q3 * q3));
}

void quaternion::normalize()
{
    double m = sqrt((q0 * q0) + (q1 * q1) + (q2 * q2) + (q3 * q3));
    if (m)
    {
    q0 = q0 / m;
    q1 = q1 / m;
    q2 = q2 / m;
    q3 = q3 / m;
```

```
    }

}

quaternion quaternion::rate_of_change(double P, double Q, double R)
{
    quaternion rate;
    rate.q0 = -0.5*((q1 * P) + (q2 * Q) + (q3 * R));
    rate.q1 = 0.5*((q0 * P) + (q2 * R) - (q3 * Q));
    rate.q2 = 0.5*((q0 * Q) + (q3 * P) - (q1 * R));
    rate.q3 = 0.5*((q0 * R) + (q1 * Q) - (q2 * P));
    return rate;

}

void quaternion::update(double P, double Q, double R, double sec)
{
//Runga Kutta fourth order method used
    double hh = sec * .5, h6 = sec / 6;
    quaternion y = *this, dym, dyt, yt, dydx;
    dydx.q0 = -0.5*((q1 * P) + (q2 * Q) + (q3 * R));
    dydx.q1 = 0.5*((q0 * P) + (q2 * R) - (q3 * Q));
    dydx.q2 = 0.5*((q0 * Q) + (q3 * P) - (q1 * R));
    dydx.q3 = 0.5*((q0 * R) + (q1 * Q) - (q2 * P));

    yt.q0 = y.q0 + hh * dydx.q0;
    yt.q1 = y.q1 + hh * dydx.q1;
    yt.q2 = y.q2 + hh * dydx.q2;
    yt.q3 = y.q3 + hh * dydx.q3;

    dyt.q0 = -0.5*((yt.q1 * P) + (yt.q2 * Q) + (yt.q3 * R));
    dyt.q1 = 0.5*((yt.q0 * P) + (yt.q2 * R) - (yt.q3 * Q));
    dyt.q2 = 0.5*((yt.q0 * Q) + (yt.q3 * P) - (yt.q1 * R));
    dyt.q3 = 0.5*((yt.q0 * R) + (yt.q1 * Q) - (yt.q2 * P));

    yt.q0 = y.q0 + hh * dyt.q0;
    yt.q1 = y.q1 + hh * dyt.q1;
    yt.q2 = y.q2 + hh * dyt.q2;
    yt.q3 = y.q3 + hh * dyt.q3;

    dym.q0 = -0.5*((yt.q1 * P) + (yt.q2 * Q) + (yt.q3 * R));
    dym.q1 = 0.5*((yt.q0 * P) + (yt.q2 * R) - (yt.q3 * Q));
    dym.q2 = 0.5*((yt.q0 * Q) + (yt.q3 * P) - (yt.q1 * R));
    dym.q3 = 0.5*((yt.q0 * R) + (yt.q1 * Q) - (yt.q2 * P));

    yt.q0 = y.q0 + sec * dym.q0;
    yt.q1 = y.q1 + sec * dym.q1;
    yt.q2 = y.q2 + sec * dym.q2;
    yt.q3 = y.q3 + sec * dym.q3;

    dym.q0 = dym.q0 + dyt.q0;
    dym.q1 = dym.q1 + dyt.q1;
    dym.q2 = dym.q2 + dyt.q2;
    dym.q3 = dym.q3 + dyt.q3;
```

```
    dyt.q0 = -0.5*((yt.q1 * P) + (yt.q2 * Q) + (yt.q3 * R));
    dyt.q1 = 0.5*((yt.q0 * P) + (yt.q2 * R) - (yt.q3 * Q));
    dyt.q2 = 0.5*((yt.q0 * Q) + (yt.q3 * P) - (yt.q1 * R));
    dyt.q3 = 0.5*((yt.q0 * R) + (yt.q1 * Q) - (yt.q2 * P));

    q0 = y.q0 + h6 * (dydx.q0 + dyt.q0 + 2.0 * dym.q0);
    q1 = y.q1 + h6 * (dydx.q1 + dyt.q1 + 2.0 * dym.q1);
    q2 = y.q2 + h6 * (dydx.q2 + dyt.q2 + 2.0 * dym.q2);
    q3 = y.q3 + h6 * (dydx.q3 + dyt.q3 + 2.0 * dym.q3);

}

quaternion quaternion::rate_of_change(vector3D ang_velocity)
{
    quaternion rate;
    rate.q0 = -0.5*((q1 * ang_velocity[0]) + (q2 * ang_velocity[1]) +
              (q3 * ang_velocity[2]));
    rate.q1 =  0.5*((q0 * ang_velocity[0]) + (q2 * ang_velocity[2]) -
              (q3 * ang_velocity[1]));
    rate.q2 =  0.5*((q0 * ang_velocity[1]) + (q3 * ang_velocity[0]) -
              (q1 * ang_velocity[2]));
    rate.q3 =  0.5*((q0 * ang_velocity[2]) + (q1 * ang_velocity[1]) -
              (q2 * ang_velocity[0]));
    return rate;

}

void quaternion::update(vector3D ang_velocity, double sec)
{
//Runga Kutta method used
    double P = ang_velocity[0], Q = ang_velocity[1], R = ang_velocity[2];
    double hh = sec * .5, h6 = sec / 6;
    quaternion y = *this, dym, dyt, yt, dydx;

    dydx.q0 = -0.5*((q1 * P) + (q2 * Q) + (q3 * R));
    dydx.q1 = 0.5*((q0 * P) + (q2 * R) - (q3 * Q));
    dydx.q2 = 0.5*((q0 * Q) + (q3 * P) - (q1 * R));
    dydx.q3 = 0.5*((q0 * R) + (q1 * Q) - (q2 * P));

    yt.q0 = y.q0 + hh * dydx.q0;
    yt.q1 = y.q1 + hh * dydx.q1;
    yt.q2 = y.q2 + hh * dydx.q2;
    yt.q3 = y.q3 + hh * dydx.q3;

    dyt.q0 = -0.5*((yt.q1 * P) + (yt.q2 * Q) + (yt.q3 * R));
    dyt.q1 = 0.5*((yt.q0 * P) + (yt.q2 * R) - (yt.q3 * Q));
    dyt.q2 = 0.5*((yt.q0 * Q) + (yt.q3 * P) - (yt.q1 * R));
    dyt.q3 = 0.5*((yt.q0 * R) + (yt.q1 * Q) - (yt.q2 * P));

    yt.q0 = y.q0 + hh * dyt.q0;
    yt.q1 = y.q1 + hh * dyt.q1;
    yt.q2 = y.q2 + hh * dyt.q2;
    yt.q3 = y.q3 + hh * dyt.q3;
```

```
    dym.q0 = -0.5*((yt.q1 * P) + (yt.q2 * Q) + (yt.q3 * R));
    dym.q1 = 0.5*((yt.q0 * P) + (yt.q2 * R) - (yt.q3 * Q));
    dym.q2 = 0.5*((yt.q0 * Q) + (yt.q3 * P) - (yt.q1 * R));
    dym.q3 = 0.5*((yt.q0 * R) + (yt.q1 * Q) - (yt.q2 * P));

    yt.q0 = y.q0 + sec * dym.q0;
    yt.q1 = y.q1 + sec * dym.q1;
    yt.q2 = y.q2 + sec * dym.q2;
    yt.q3 = y.q3 + sec * dym.q3;

    dym.q0 = dym.q0 + dyt.q0;
    dym.q1 = dym.q1 + dyt.q1;
    dym.q2 = dym.q2 + dyt.q2;
    dym.q3 = dym.q3 + dyt.q3;

    dyt.q0 = -0.5*((yt.q1 * P) + (yt.q2 * Q) + (yt.q3 * R));
    dyt.q1 = 0.5*((yt.q0 * P) + (yt.q2 * R) - (yt.q3 * Q));
    dyt.q2 = 0.5*((yt.q0 * Q) + (yt.q3 * P) - (yt.q1 * R));
    dyt.q3 = 0.5*((yt.q0 * R) + (yt.q1 * Q) - (yt.q2 * P));

    q0 = y.q0 + h6 * (dydx.q0 + dyt.q0 + 2.0 * dym.q0);
    q1 = y.q1 + h6 * (dydx.q1 + dyt.q1 + 2.0 * dym.q1);
    q2 = y.q2 + h6 * (dydx.q2 + dyt.q2 + 2.0 * dym.q2);
    q3 = y.q3 + h6 * (dydx.q3 + dyt.q3 + 2.0 * dym.q3);
}

ostream& operator<<(ostream& os, quaternion& q)
{
    os << (double) q.q0 << ", " << (double) q.q1 << ", " << (double) q.q2 << ", " << (double) q.q3 <<
"\n";
    return os;
}

double& quaternion::operator[](int n)
{
    if (n == 0)
    {
        return q0;
    }
    if (n == 1)
    {
        return q1;
    }
    if (n == 2)
    {
        return q2;
    }
    if (n == 3)
    {
        return q3;
    }
}
#endif
```

APPENDIX G

Matrix3x3 Code

A. HEADER FILE

```
#ifndef MATRIX3X3_H
#define MATRIX3X3_H
#include "vector3D.H"
#include "quaternion.H"
#include <iostream.h>

class matrix3x3
{
    double m[9];
public:
    matrix3x3();
    matrix3x3(double, double, double, double, double, double, double, double, double);
    matrix3x3(const matrix3x3&);
    matrix3x3& operator=(const matrix3x3&);
    matrix3x3 operator+(const matrix3x3&);
    matrix3x3 operator-(const matrix3x3&);
    matrix3x3 operator*(const matrix3x3&);
    void DCM_x_rotation(double);
    void DCM_y_rotation(double);
    void DCM_z_rotation(double);
    void DCM_body_to_world(quaternion);
    void DCM_world_to_body(quaternion);
    void transpose();
    quaternion generate_orientation();
    vector3D operator*(vector3D&);
    matrix3x3 operator*(double);
    double& operator[](int);
    friend ostream& operator<<(ostream&, matrix3x3&);
    ~matrix3x3() { }
};
#endif
```

B. SOURCE FILE

```
#ifndef MATRIX3X3_C
#define MATRIX3X3_C
#include "matrix3x3.H"


matrix3x3::matrix3x3()
{
    m[0] = 1;
    m[1] = 0;
    m[2] = 0;
    m[3] = 0;
    m[4] = 1;
    m[5] = 0;
    m[6] = 0;
    m[7] = 0;
    m[8] = 1;

}

matrix3x3::matrix3x3(double a, double b, double c, double d, double e, double f, double g, double h,
double i)
{
    m[0] = a;
    m[1] = b;
    m[2] = c;
    m[3] = d;
    m[4] = e;
    m[5] = f;
    m[6] = g;
    m[7] = h;
    m[8] = i;
}

matrix3x3::matrix3x3(const matrix3x3& v)
{
    m[0] = v.m[0];
    m[1] = v.m[1];
    m[2] = v.m[2];
    m[3] = v.m[3];
    m[4] = v.m[4];
    m[5] = v.m[5];
    m[6] = v.m[6];
    m[7] = v.m[7];
    m[8] = v.m[8];
}

matrix3x3& matrix3x3::operator=(const matrix3x3& v)
{
    m[0] = v.m[0];
    m[1] = v.m[1];
    m[2] = v.m[2];
    m[3] = v.m[3];
```

```
    m[4] = v.m[4];
    m[5] = v.m[5];
    m[6] = v.m[6];
    m[7] = v.m[7];
    m[8] = v.m[8];
    return *this;
}

matrix3x3 matrix3x3::operator+(const matrix3x3& v)
{
    matrix3x3 sum;
    sum.m[0] = m[0] + v.m[0];
    sum.m[1] = m[1] + v.m[1];
    sum.m[2] = m[2] + v.m[2];
    sum.m[3] = m[3] + v.m[3];
    sum.m[4] = m[4] + v.m[4];
    sum.m[5] = m[5] + v.m[5];
    sum.m[6] = m[6] + v.m[6];
    sum.m[7] = m[7] + v.m[7];
    sum.m[8] = m[8] + v.m[8];
    return sum;
}

matrix3x3 matrix3x3::operator-(const matrix3x3& v)
{
    matrix3x3 difference;
    difference.m[0] = m[0] - v.m[0];
    difference.m[1] = m[1] - v.m[1];
    difference.m[2] = m[2] - v.m[2];
    difference.m[3] = m[3] - v.m[3];
    difference.m[4] = m[4] - v.m[4];
    difference.m[5] = m[5] - v.m[5];
    difference.m[6] = m[6] - v.m[6];
    difference.m[7] = m[7] - v.m[7];
    difference.m[8] = m[8] - v.m[8];
    return difference;
}

//matrix multiplication
matrix3x3 matrix3x3::operator*(const matrix3x3& v)
{
    matrix3x3 temp;
    temp.m[0] = (m[0] * v.m[0]) + (m[1] * v.m[3]) + (m[2] * v.m[6]);
    temp.m[1] = (m[0] * v.m[1]) + (m[1] * v.m[4]) + (m[2] * v.m[7]);
    temp.m[2] = (m[0] * v.m[2]) + (m[1] * v.m[5]) + (m[2] * v.m[8]);
    temp.m[3] = (m[3] * v.m[0]) + (m[4] * v.m[3]) + (m[5] * v.m[6]);
    temp.m[4] = (m[3] * v.m[1]) + (m[4] * v.m[4]) + (m[5] * v.m[7]);
    temp.m[5] = (m[3] * v.m[2]) + (m[4] * v.m[5]) + (m[5] * v.m[8]);
    temp.m[6] = (m[6] * v.m[0]) + (m[7] * v.m[3]) + (m[8] * v.m[6]);
    temp.m[7] = (m[6] * v.m[1]) + (m[7] * v.m[4]) + (m[8] * v.m[7]);
    temp.m[8] = (m[6] * v.m[2]) + (m[7] * v.m[5]) + (m[8] * v.m[8]);
    return temp;

}
```

```
//matrix multiplication with a vector
vector3D matrix3x3::operator*(vector3D& v)
{
    vector3D temp = v;
    temp[0] = (m[0] * v[0]) + (m[1] * v[1]) + (m[2] * v[2]);
    temp[1] = (m[3] * v[0]) + (m[4] * v[1]) + (m[5] * v[2]);
    temp[2] = (m[6] * v[0]) + (m[7] * v[1]) + (m[8] * v[2]);
    return temp;
}

//scalar multiplication
matrix3x3 matrix3x3::operator*(double n)
{
    matrix3x3 product;
    product.m[0] = m[0] * n;
    product.m[1] = m[1] * n;
    product.m[2] = m[2] * n;
    product.m[3] = m[3] * n;
    product.m[4] = m[4] * n;
    product.m[5] = m[5] * n;
    product.m[6] = m[6] * n;
    product.m[7] = m[7] * n;
    product.m[8] = m[8] * n;
    return product;
}

ostream& operator<<(ostream& os, matrix3x3& v)
{
    os << (double) v.m[0] << ", " << (double) v.m[1] << ", " << (double) v.m[2] << "\n"
      << (double) v.m[3] << ", " << (double) v.m[4] << ", " << (double) v.m[5] << "\n"
      << (double) v.m[6] << ", " << (double) v.m[7] << ", " << (double) v.m[8] << "\n" << "\n";
    return os;
}

double& matrix3x3::operator[](int y)
{
    return m[y];
}

//generates DCM for rotation about the x axis
void matrix3x3::DCM_x_rotation(double angle)
{
    m[0] = 1.0;
    m[1] = 0.0;
    m[2] = 0.0;
    m[3] = 0.0;
    m[4] = cos(angle);
    m[5] = sin(angle);
    m[6] = 0.0;
    m[7] = -sin(angle);
    m[8] = cos(angle);
}
```

```
//generates DCM for rotation about the y axis
void matrix3x3::DCM_y_rotation(double angle)
{
    m[0] = cos(angle);
    m[1] = 0.0;
    m[2] = -sin(angle);
    m[3] = 0.0;
    m[4] = 1.0;
    m[5] = 0.0;
    m[6] = sin(angle);
    m[7] = 0.0;
    m[8] = cos(angle);
}

//generates DCM for rotation about the z axis
void matrix3x3::DCM_z_rotation(double angle)
{
    m[0] = cos(angle);
    m[1] = sin(angle);
    m[2] = 0.0;
    m[3] = -sin(angle);
    m[4] = cos(angle);
    m[5] = 0.0;
    m[6] = 0.0;
    m[7] = 0.0;
    m[8] = 1.0;
}

//generates DCM for transformation from body to world coordinates
void matrix3x3::DCM_body_to_world(quaternion orientation)
{
m[0] = ((orientation[0] * orientation[0]) + (orientation[1] *
        orientation[1]) - (orientation[2] * orientation[2]) -
       (orientation[3] * orientation[3]));
m[1] = 2.0 * ((orientation[1] * orientation[2]) - (orientation[0] *
             orientation[3]));
m[2] = 2.0 * ((orientation[0] * orientation[2]) + (orientation[1] *
             orientation[3]));
m[3] = 2.0 * ((orientation[1] * orientation[2]) + (orientation[0] *
             orientation[3]));
m[4] = ((orientation[0] * orientation[0]) - (orientation[1] *
        orientation[1]) + (orientation[2] * orientation[2]) -
       (orientation[3] * orientation[3]));
m[5] = 2.0 * ((orientation[2] * orientation[3]) - (orientation[0] *
             orientation[1]));
m[6] = 2.0 * ((orientation[1] * orientation[3]) - (orientation[0] *
             orientation[2]));
m[7] = 2.0 * ((orientation[2] * orientation[3]) + (orientation[0] *
             orientation[1]));
m[8] = ((orientation[0] * orientation[0]) - (orientation[1] *
        orientation[1]) - (orientation[2] * orientation[2]) +
       (orientation[3] * orientation[3]));
}
```

```
//generates DCM for transformation from world to body coordinates
void matrix3x3::DCM_world_to_body(quaternion orientation)
{
m[0] = ((orientation[0] * orientation[0]) + (orientation[1] *
        orientation[1]) - (orientation[2] * orientation[2]) -
       (orientation[3] * orientation[3]));
m[3] = 2.0 * ((orientation[1] * orientation[2]) - (orientation[0] *
             orientation[3]));
m[6] = 2.0 * ((orientation[0] * orientation[2]) + (orientation[1] *
             orientation[3]));
m[1] = 2.0 * ((orientation[1] * orientation[2]) + (orientation[0] *
             orientation[3]));
m[4] = ((orientation[0] * orientation[0]) - (orientation[1] *
        orientation[1]) + (orientation[2] * orientation[2]) -
       (orientation[3] * orientation[3]));
m[7] = 2.0 * ((orientation[2] * orientation[3]) - (orientation[0] *
             orientation[1]));
m[2] = 2.0 * ((orientation[1] * orientation[3]) - (orientation[0] *
             orientation[2]));
m[5] = 2.0 * ((orientation[2] * orientation[3]) + (orientation[0] *
             orientation[1]));
m[8] = ((orientation[0] * orientation[0]) - (orientation[1] *
        orientation[1]) - (orientation[2] * orientation[2]) +
       (orientation[3] * orientation[3]));
}

void matrix3x3::transpose()
{
    matrix3x3 v = *this;
    m[0] = v.m[0];
    m[1] = v.m[3];
    m[2] = v.m[6];
    m[3] = v.m[1];
    m[4] = v.m[4];
    m[5] = v.m[7];
    m[6] = v.m[2];
    m[7] = v.m[5];
    m[8] = v.m[8];
}

quaternion matrix3x3::generate_orientation()
{
    quaternion q;
    q[0] = 0.5 * sqrt(1 + m[0] + m[4] + m[8]);
    q[1] = 0.5 * sqrt(1 + m[0] - m[4] - m[8]);
    q[2] = 0.5 * sqrt(1 - m[0] + m[4] - m[8]);
    q[3] = 0.5 * sqrt(1 - m[0] - m[4] + m[8]);
    q.normalize();
    return q;
}

#endif
```

APPENDIX H

Time Code

A. HEADER FILE

```
#ifndef TIME_H
#define TIME_H
#include <time.h>
#include <sys/types.h>
#include <sys/times.h>
#include <sys/param.h>


void set_delta();
void set_delta(double);
void set_time();
void reset_time();
double read_delta();
double read_time();
int read_ticks();
void set_real_time_factor(double);
#endif
```

B. SOURCE FILE

```
#ifndef TIME_C
#define TIME_C
#include "time.H"

struct tms timebuffer;
long old_time;
double delta, real_time_ratio = 1.0;

void set_delta()
{
// HZ is the system clock rate in hertz
    delta = ((double) (times(&timebuffer) - old_time)/HZ) * real_time_ratio;
    old_time = times(&timebuffer);
}
void set_delta(double step)
{
    delta = step;
    old_time = times(&timebuffer);
}

void set_time()
{
    old_time = times(&timebuffer);
}

double read_delta()
{
    return delta;
}

double read_time()
{
    return (double) old_time;
}

int read_ticks()
{
    return (int) times(&timebuffer);
}

void set_real_time_factor(double f)
{
    real_time_ratio = f;
}

void reset_time()
{
    long delta_ticks;
    delta_ticks = (long) (delta * HZ);
    old_time = times(&timebuffer) - delta_ticks;
}
#endif
```

APPENDIX I

Graphics Code

A. HEADER FILE

```
#ifndef GRAPHICS_H
#define GRAPHICS_H
#include <math.h>
#include "vector3D.H"
#include "matrix3x3.H"
#include "quaternion.H"
#include "rdobj_opcodes.h"
#include "rdobj_funcs.h"
#include <stdio.h>
#include <gl.h>
#include <device.h>

//initializes the graphic system
void initialize();

//initializes control window
void init_control_window();

 //make viewing window active
void main_window();

// makes control window active
void control_window();

//clears control window
void clear_control_window();

//control window for euler program
void euler_controls(int, int, int, int, int, int, int,quaternion, double);

//control window for gyro program
void gyro_controls(int, int, int, int, vector3D, int, int, int, double, double, double, double);

//statisic display for gyro program
void stat_controls(double, double, double, double, double, double, double, double, vector3D*);

//control window for frame program
void frame_controls(int, int, vector3D, vector3D, vector3D, int);

//standard function for viewing a scene
void view();

//used to view the scene for a point of view fixed to a rigid body
void view(quaternion, vector3D, int);
```

```
//attaches the eye to a rigid body
void attach_eye_to(vector3D*, int*);

//attaches the tatget to a rigid body
void attach_target_to(vector3D*);

//attaches the eye to a rigid body
void set_eye_to(double, double, double);

//attaches the target to a rigid body
void set_target_to(double, double, double);

//rotates the view in tenths of degrees
void rotate_view(int);

//displays the body axes of a rigid_body
void view_axis();

//gravity check - returns non zero value when gravity is turned on
int gravity_status();
void set_gravity_on();
void set_gravity_off();
void toggle_gravity();
//air resistance check - returns non zero value when air resistance is turned on
int air_resistance_status();
void set_air_resistance_on();
void set_air_resistance_off();
void toggle_air_resistance();

// c routines the must be accessed
extern "C"
{
    extern OBJECT* read_object(char[]);
    extern void ready_object_for_display( OBJECT* );
    extern void display_this_object( OBJECT* );
};
#endif
```

B. SOURCE FILE

```
#ifndef GRAPHICS_C
#define GRAPHICS_C
#include "graphics.H"
#define NEARDEPTH 0x000000 /* the near and far planes used for Zbuffering*/
#define FARDEPTH  0x7fffff

OBJECT *lightobj, *axis;
//eye and target are the global variables that control the view point
//and reference point of the scene respectively
vector3D *eye = new vector3D(10.0, 10.0, 10.0), *target = new vector3D;
int *eye_display_field = NULL;
int gravity_flag = 0, air_resistance_flag = 0;
int twist = 0;
long main_win, control_win;

Matrix un = { 1.0, 0.0, 0.0, 0.0,
        0.0, 1.0, 0.0, 0.0,
        0.0, 0.0, 1.0, 0.0,
        0.0, 0.0, 0.0, 1.0};

int gravity_status()
{
    return gravity_flag;
}

void set_gravity_on()
{
    gravity_flag = 1;
}

void set_gravity_off()
{
    gravity_flag = 0;
}

void toggle_gravity()
{
    if(gravity_flag)
    {
        gravity_flag = 0;
    }
    else
    {
        gravity_flag = 1;
    }
}

int air_resistance_status()
{
    return air_resistance_flag;
}
```

```
void set_air_resistance_on()
{
    air_resistance_flag = 1;
}

void set_air_resistance_off()
{
    air_resistance_flag = 0;
}

void toggle_air_resistance()
{
    if(air_resistance_flag)
    {
        air_resistance_flag = 0;
    }
    else
    {
        air_resistance_flag = 1;
    }
}


void initialize()
{

  /* set up the preferred aspect ratio */
  keepaspect(XMAXSCREEN+1,YMAXSCREEN+1);
  prefsize(XMAXSCREEN/2,YMAXSCREEN/2);
  prefposition(0,XMAXSCREEN * 0.8 ,0,YMAXSCREEN * 0.8);
  /* open a window for the program */
  main_win = winopen("Main");
  wintitle("Physically Based Reality, A Keith Haynes Production");
  /* put the IRIS into double buffer mode */
  doublebuffer();
  /* put the iris into rgb mode */
  RGBmode();
  /* configure the IRIS (means use the above command settings) */
  gconfig();
  /* set the depth for z-buffering */
  lsetdepth(NEARDEPTH,FARDEPTH);
  /* queue the redraw device */
  qdevice(REDRAW);
  /* queue the menubutton */
  qdevice(MENUBUTTON);
  /* turn the cursor on */
  curson();
  /* select gouraud shading */
  shademodel(GOURAUD);
  /* turn on the new projection matrix mode */
  mmode(MVIEWING);
  /*Turn of Zbuffering*/
  zbuffer(TRUE);
  lightobj = read_object("the_light.off");
```

```
  axis = read_object("frame.off");
  ready_object_for_display(lightobj);
  ready_object_for_display(axis);
  //queue up input devices
  qdevice(LEFTMOUSE);
  qdevice(RIGHTMOUSE);
  qdevice(MOUSEX);
  qdevice(MOUSEY);
  qdevice(RIGHTARROWKEY);
  qdevice(LEFTARROWKEY);
  qdevice(UPARROWKEY);
  qdevice(DOWNARROWKEY);
  qdevice(SPACEKEY);
  qdevice(EQUALKEY);
  qdevice(MINUSKEY);
  qdevice(F1KEY);
  qdevice(F2KEY);
  qdevice(F3KEY);
  qdevice(F4KEY);
  qdevice(F5KEY);
  qdevice(F6KEY);
  qdevice(F7KEY);
  qdevice(F8KEY);
  qdevice(F9KEY);
  qdevice(F10KEY);
  qdevice(F11KEY);
  qdevice(F12KEY);
  //clear draw and display buffer
  czclear(0xFFFF7200,getgdesc(GD_ZMAX));
  swapbuffers();
  czclear(0xFFFF7200,getgdesc(GD_ZMAX));
}

void init_control_window()
{
  /* set up the preferred aspect ratio */
  prefposition(0,XMAXSCREEN * 0.8,YMAXSCREEN * 0.87, YMAXSCREEN);
  /* open a window for the program */
  control_win = winopen("control");
  wintitle("System Control Window");
  /* put the IRIS into double buffer mode */
  doublebuffer();
  RGBmode();
  gconfig();
  pushmatrix();
  ortho2(0.0, 769.0, 0.0, 100.0);
  RGBcolor(255,255,255);
  clear();
  popmatrix();
  swapbuffers();
}

void main_window()
{
```

```
    winset(main_win);
}

void control_window()
{
    winset(control_win);
}

void clear_control_window()
{
    winset(control_win);
    pushmatrix();
    ortho2(0.0, 769.0, 0.0, 100.0);
    RGBcolor(255,255,255);
    clear();
    popmatrix();
    swapbuffers();
    winset(main_win);
}

void euler_controls(int ang1, int ang2, int ang3, int axis1, int axis2, int axis3, int q, quaternion
orientation, double theta)
{
//ang1, ang2, ang3 are the amounts of the rotations
//axis1, axis2, axis3 are the about which the shuttle is rotated
//q is the flag for placing an X in the quaternion equivilant box
//orientation is the quaternion from the shuttle
//theta is the value for the rotation using a single quaternion rotation

winset(control_win);
//up and down arrows
float pt [3][2] = {
        {202,58},
        {208,58},
        {205,52}};
float pt2 [3][2] = {
        {142,52},
        {148,52},
        {145,58}};
char s[32];
pushmatrix();
ortho2(0.0, 769.0, 0.0, 100.0);
RGBcolor(255,255,255);
clear();
//set axis color for rotation 1
switch(axis1)
{
    case 1:
    RGBcolor(200,0,0);
    break;

    case 2:
    RGBcolor(0,0,200);
    break;
```

```
        case 3:
        RGBcolor(0,0,0);
        break;

        default:
        RGBcolor(200,0,0);
        break;
}
//draw boxes for first rotation angle and axis
rectf(150.0, 50.0, 200.0, 60.0);
rectf(170.0, 35.0, 180.0, 45.0);

//set axis color for rotation 2
switch(axis2)
{
        case 1:
        RGBcolor(200,0,0);
        break;

        case 2:
        RGBcolor(0,0,200);
        break;

        case 3:
        RGBcolor(0,0,0);
        break;

        default:
        RGBcolor(0,0,200);
        break;
}
//draw boxes for second rotation angle and axis
rectf(250.0, 50.0, 300.0, 60.0);
rectf(270.0, 35.0, 280.0, 45.0);

//set axis color for rotation 3
switch(axis3)
{
        case 1:
        RGBcolor(200,0,0);
        break;

        case 2:
        RGBcolor(0,0,200);
        break;

        case 3:
        RGBcolor(0,0,0);
        break;

        default:
        RGBcolor(0,0,0);
        break;
```

```
}
//draw boxes for third rotation angle and axis
rectf(350.0, 50.0, 400.0, 60.0);
rectf(370.0, 35.0, 380.0, 45.0);

RGBcolor(200,200,200);
//area for up and down arrows
rectf(140.0, 50.0, 150.0, 60.0);
rectf(200.0, 50.0, 210.0, 60.0);
rectf(240.0, 50.0, 250.0, 60.0);
rectf(300.0, 50.0, 310.0, 60.0);
rectf(340.0, 50.0, 350.0, 60.0);
rectf(400.0, 50.0, 410.0, 60.0);

//reset and go buttons
rectf(550.0, 25.0, 600.0, 75.0);
rectf(625.0, 25.0, 675.0, 75.0);
RGBcolor(0,0,0);
// Draw up and down arrows
polf2(3,pt);
polf2(3,pt2);
pt[0][0] = pt[0][0] + 100.0;
pt[1][0] = pt[1][0] + 100.0;
pt[2][0] = pt[2][0] + 100.0;
pt2[0][0] = pt2[0][0] + 100.0;
pt2[1][0] = pt2[1][0] + 100.0;
pt2[2][0] = pt2[2][0] + 100.0;
polf2(3,pt);
polf2(3,pt2);
pt[0][0] = pt[0][0] + 100.0;
pt[1][0] = pt[1][0] + 100.0;
pt[2][0] = pt[2][0] + 100.0;
pt2[0][0] = pt2[0][0] + 100.0;
pt2[1][0] = pt2[1][0] + 100.0;
pt2[2][0] = pt2[2][0] + 100.0;
polf2(3,pt);
polf2(3,pt2);

//Show Quaternion;
RGBcolor(0,0,255);
rectf(190.0, 15.0, 200.0, 25.0);
RGBcolor(255,255,0);
if(q)
{
    cmov2(193.5,17.0);
    charstr("X");
}

//Rotation Output
RGBcolor(255,0,0);
rectf(420.0, 10.0, 520.0, 80.0);
RGBcolor(0,0,0);
cmov2(425.0,66.0);
charstr("Theta =");
```

```
cmov2(425.0,51.0);
charstr("q0 =");
cmov2(425.0,41.0);
charstr("q1 =");
cmov2(425.0,31.0);
charstr("q2 =");
cmov2(425.0,21.0);
charstr("q3 =");
RGBcolor(255,255,255);
cmov2(475, 66);
sprintf(s, "%.1f", theta * 57.29578);
charstr(s);
cmov2(455, 51);
sprintf(s, "%.4f", orientation[0]);
charstr(s);
cmov2(455, 41);
sprintf(s, "%.4f", orientation[1]);
charstr(s);
cmov2(455, 31);
sprintf(s, "%.4f", orientation[2]);
charstr(s);
cmov2(455, 21);
sprintf(s, "%.4f", orientation[3]);
charstr(s);



//Control Window text
RGBcolor(0,0,0);
cmov2(10.0,90.0);
charstr("Euler Rotation Control Window");
cmov2(10,50);
charstr("Rotation in Degrees");
cmov2(10,35);
charstr("Axis of Rotation");
cmov2(10,15);
charstr("Show Quaternion Equivalent");
cmov2(570, 45);
charstr("GO");
cmov2(635,45);
charstr("Reset");

//angles and axis output
RGBcolor(255,255,0);
cmov2(160, 51);
sprintf(s, "%.0f", (double) ang1);
charstr(s);
cmov2(260, 51);
sprintf(s, "%.0f", (double) ang2);
charstr(s);
cmov2(360, 51);
sprintf(s, "%.0f", (double) ang3);
charstr(s);
cmov2(172, 36);
```

```
sprintf(s, "%.0f", (double) axis1);
charstr(s);
cmov2(272, 36);
sprintf(s, "%.0f", (double) axis2);
charstr(s);
cmov2(372, 36);
sprintf(s, "%.0f", (double) axis3);
charstr(s);

popmatrix();
swapbuffers();
}

.

void frame_controls(int vlevel, int flevel, vector3D mag, vector3D pos, vector3D av, int vaxis)
{
//vlevel - viewing level, flevel - assignment level, mag - linear velocity, pos - position
//av angular velocity, vaxis - viewing axis
float pt [3][2] = {
        {192,48},
        {198,48},
        {195,42}};
float pt2 [3][2] = {
        {232,42},
        {238,42},
        {235,48}};
winset(control_win);
char s[32];
pushmatrix();
ortho2(0.0, 769.0, 0.0, 100.0);
RGBcolor(255,255,255);
clear();

//Go & Reset Buttons
RGBcolor(200,200,200);
rectf(550.0, 25.0, 600.0, 75.0);
rectf(625.0, 25.0, 675.0, 75.0);
RGBcolor(0,0,0);
cmov2(570, 45);
charstr("GO");
cmov2(635,45);
charstr("Reset");

// Select viewing level
RGBcolor(0,0,255);
rectf(10.0, 0.0, 65.0, 60.0);
RGBcolor(255,255,0);
cmov2(11.0,51.5);
charstr("Inertial");
cmov2(35.0,41.5);
charstr("1");
cmov2(35.0,31.5);
charstr("2");
cmov2(35.0,21.5);
```

```
charstr("3");
cmov2(35.0,11.5);
charstr("4");
cmov2(35.0,1.5);
charstr("5");

switch(vlevel)
{
    case 0:
    rectf(10.0, 50.0, 65.0, 60.0);
    RGBcolor(0,0,255);
    cmov2(11.0,51.5);
    charstr("Inertial");
    break;

    case 1:
    rectf(10.0, 40.0, 65.0, 50.0);
    RGBcolor(0,0,255);
    cmov2(35.0,41.5);
    charstr("1");
    break;

    case 2:
    rectf(10.0, 30.0, 65.0, 40.0);
    RGBcolor(0,0,255);
    cmov2(35.0,31.5);
    charstr("2");
    break;

    case 3:
    rectf(10.0, 20.0, 65.0, 30.0);
    RGBcolor(0,0,255);
    cmov2(35.0,21.5);
    charstr("3");
    break;

    case 4:
    rectf(10.0, 10.0, 65.0, 20.0);
    RGBcolor(0,0,255);
    cmov2(35.0,11.5);
    charstr("4");
    break;

    case 5:
    rectf(10.0, 0.0, 65.0, 10.0);
    RGBcolor(0,0,255);
    cmov2(35.0,1.5);
    charstr("5");
    break;

    default:
    rectf(10.0, 50.0, 65.0, 60.0);
    RGBcolor(0,0,255);
    cmov2(11.0,51.5);
```

```
        charstr("Inertial");
        break;

}

// Select frame assignment level
RGBcolor(0,0,255);
rectf(70.0, 0.0, 110.0, 50.0);
RGBcolor(255,255,0);
cmov2(88.0,41.5);
charstr("1");
cmov2(88.0,31.5);
charstr("2");
cmov2(88.0,21.5);
charstr("3");
cmov2(88.0,11.5);
charstr("4");
cmov2(88.0,1.5);
charstr("5");

switch(flevel)
{
        case 1:
        rectf(70.0, 40.0, 110.0, 50.0);
        RGBcolor(0,0,255);
        cmov2(88.0,41.5);
        charstr("1");
        break;

        case 2:
        rectf(70.0, 30.0, 110.0, 40.0);
        RGBcolor(0,0,255);
        cmov2(88.0,31.5);
        charstr("2");
        break;

        case 3:
        rectf(70.0, 20.0, 110.0, 30.0);
        RGBcolor(0,0,255);
        cmov2(88.0,21.5);
        charstr("3");
        break;

        case 4:
        rectf(70.0, 10.0, 110.0, 20.0);
        RGBcolor(0,0,255);
        cmov2(88.0,11.5);
        charstr("4");
        break;

        case 5:
        rectf(70.0, 0.0, 110.0, 10.0);
        RGBcolor(0,0,255);
        cmov2(88.0,1.5);
```

```
        charstr("5");
        break;

        default:
        rectf(70.0, 40.0, 110.0, 50.0);
        RGBcolor(0,0,255);
        cmov2(88.0,41.5);
        charstr("1");
        break;
}

// Velocity
RGBcolor(0,0,255);
rectf(200.0, 20.0, 230.0, 50.0);
RGBcolor(200,200,200);
rectf(190.0, 20.0, 200.0, 50.0);
rectf(230.0, 20.0, 240.0, 50.0);
// Draw up and down arrows
RGBcolor(0,0,0);
polf2(3,pt);
polf2(3,pt2);
pt[0][1] = pt[0][1] - 10.0;
pt[1][1] = pt[1][1] - 10.0;
pt[2][1] = pt[2][1] - 10.0;
pt2[0][1] = pt2[0][1] - 10.0;
pt2[1][1] = pt2[1][1] - 10.0;
pt2[2][1] = pt2[2][1] - 10.0;
polf2(3,pt);
polf2(3,pt2);
pt[0][1] = pt[0][1] - 10.0;
pt[1][1] = pt[1][1] - 10.0;
pt[2][1] = pt[2][1] - 10.0;
pt2[0][1] = pt2[0][1] - 10.0;
pt2[1][1] = pt2[1][1] - 10.0;
pt2[2][1] = pt2[2][1] - 10.0;
polf2(3,pt);
polf2(3,pt2);
RGBcolor(255,255,0);
cmov2(202, 41);
sprintf(s, "%.1f", mag[0]);
charstr(s);
cmov2(202, 31);
sprintf(s, "%.1f", mag[1]);
charstr(s);
cmov2(202, 21);
sprintf(s, "%.1f", mag[2]);
charstr(s);


// Position
RGBcolor(0,0,255);
rectf(265.0, 20.0, 295.0, 50.0);
RGBcolor(200,200,200);
rectf(255.0, 20.0, 265.0, 50.0);
```

```
rectf(295.0, 20.0, 305.0, 50.0);
// Draw up and down arrows
RGBcolor(0,0,0);
pt[0][0] = pt[0][0] + 65.0;
pt[1][0] = pt[1][0] + 65.0;
pt[2][0] = pt[2][0] + 65.0;
pt2[0][0] = pt2[0][0] + 65.0;
pt2[1][0] = pt2[1][0] + 65.0;
pt2[2][0] = pt2[2][0] + 65.0;
polf2(3,pt);
polf2(3,pt2);
pt[0][1] = pt[0][1] + 10.0;
pt[1][1] = pt[1][1] + 10.0;
pt[2][1] = pt[2][1] + 10.0;
pt2[0][1] = pt2[0][1] + 10.0;
pt2[1][1] = pt2[1][1] + 10.0;
pt2[2][1] = pt2[2][1] + 10.0;
polf2(3,pt);
polf2(3,pt2);
pt[0][1] = pt[0][1] + 10.0;
pt[1][1] = pt[1][1] + 10.0;
pt[2][1] = pt[2][1] + 10.0;
pt2[0][1] = pt2[0][1] + 10.0;
pt2[1][1] = pt2[1][1] + 10.0;
pt2[2][1] = pt2[2][1] + 10.0;
polf2(3,pt);
polf2(3,pt2);
RGBcolor(255,255,0);
cmov2(267, 41);
sprintf(s, "%.1f", pos[0]);
charstr(s);
cmov2(267, 31);
sprintf(s, "%.1f", pos[1]);
charstr(s);
cmov2(267, 21);
sprintf(s, "%.1f", pos[2]);
charstr(s);

// Angular Velocity
RGBcolor(0,0,255);
rectf(330.0, 20.0, 360.0, 50.0);
RGBcolor(200,200,200);
rectf(320.0, 20.0, 330.0, 50.0);
rectf(360.0, 20.0, 370.0, 50.0);
// Draw up and down arrows
RGBcolor(0,0,0);
pt[0][0] = pt[0][0] + 65.0;
pt[1][0] = pt[1][0] + 65.0;
pt[2][0] = pt[2][0] + 65.0;
pt2[0][0] = pt2[0][0] + 65.0;
pt2[1][0] = pt2[1][0] + 65.0;
pt2[2][0] = pt2[2][0] + 65.0;
polf2(3,pt);
polf2(3,pt2);
```

```
pt[0][1] = pt[0][1] - 10.0;
pt[1][1] = pt[1][1] - 10.0;
pt[2][1] = pt[2][1] - 10.0;
pt2[0][1] = pt2[0][1] - 10.0;
pt2[1][1] = pt2[1][1] - 10.0;
pt2[2][1] = pt2[2][1] - 10.0;
polf2(3,pt);
polf2(3,pt2);
pt[0][1] = pt[0][1] - 10.0;
pt[1][1] = pt[1][1] - 10.0;
pt[2][1] = pt[2][1] - 10.0;
pt2[0][1] = pt2[0][1] - 10.0;
pt2[1][1] = pt2[1][1] - 10.0;
pt2[2][1] = pt2[2][1] - 10.0;
polf2(3,pt);
polf2(3,pt2);
RGBcolor(255,255,0);
cmov2(332, 41);
sprintf(s, "%.1f", av[0]);
charstr(s);
cmov2(332, 31);
sprintf(s, "%.1f", av[1]);
charstr(s);
cmov2(332, 21);
sprintf(s, "%.1f", av[2]);
charstr(s);

//window text
RGBcolor(0,0,0);
cmov2(10.0,90.0);
charstr("Particle Dynamics Control Window");
cmov2(20.0,75.0);
charstr("View");
cmov2(20.0,65.0);
charstr("Level");
cmov2(75.0,65.0);
charstr("Frame");
cmov2(75.0,55.0);
charstr("Level");
//cmov2(125.0,55.0);
//charstr("Motion");
cmov2(185.0,65.0);
charstr("Linear");
cmov2(185.0,55.0);
charstr("Velocity");
cmov2(255.0,55.0);
charstr("Position");
cmov2(320.0,65.0);
charstr("Angular");
cmov2(320.0,55.0);
charstr("Velocity");

if(vlevel == 0)  //viewing axes are only display if viewing level is not inertial
{
```

```
}
else
{
    RGBcolor(0,0,0);
    cmov2(390.0,65.0);
    charstr("View");
    cmov2(390.0,55.0);
    charstr("Axis");
    RGBcolor(0,0,255);
    rectf(390.0, 20.0, 430.0, 50.0);
    RGBcolor(255,255,0);
    cmov2(394.0,41.0);
    charstr("-X");
    cmov2(414.0,41.0);
    charstr("+X");
    cmov2(394.0,31.0);
    charstr("-Y");
    cmov2(414.0,31.0);
    charstr("+Y");
    cmov2(394.0,21.0);
    charstr("-Z");
    cmov2(414.0,21.0);
    charstr("+Z");

    switch(vaxis)
    {
        case -3:
        rectf(390.0, 20.0, 410.0, 30.0);
        RGBcolor(0,0,255);
        cmov2(394.0,21.0);
        charstr("-Z");
        break;

        case -2:
        rectf(390.0, 30.0, 410.0, 40.0);
        RGBcolor(0,0,255);
        cmov2(394.0,31.0);
        charstr("-Y");
        break;

        case -1:
        rectf(390.0, 40.0, 410.0, 50.0);
        RGBcolor(0,0,255);
        cmov2(394.0,41.0);
        charstr("-X");
        break;

        case 1:
        rectf(410.0, 40.0, 430.0, 50.0);
        RGBcolor(0,0,255);
        cmov2(414.0,41.0);
        charstr("+X");
        break;
```

```
            case 2:
            rectf(410.0, 30.0, 430.0, 40.0);
            RGBcolor(0,0,255);
            cmov2(414.0,31.0);
            charstr("+Y");
            break;

            case 3:
            rectf(410.0, 20.0, 430.0, 30.0);
            RGBcolor(0,0,255);
            cmov2(414.0,21.0);
            charstr("+Z");
            break;

            default:
            rectf(390.0, 40.0, 410.0, 50.0);
            RGBcolor(0,0,255);
            cmov2(394.0,41.0);
            charstr("-X");
            break;
        }
}
popmatrix();
swapbuffers();
}


void gyro_controls(int x, int y, int z, int object, vector3D size, int t1, int t2, int t3, double duration,
                        double mag, double elapsed, double total)
{
//x, y, z are assigned angular velocities.  object is the body type,
//t1, t2, t3 are the applied moments
float pt [3][2] = {
            {142,48},
            {148,48},
            {145,42}};
float pt2 [3][2] = {
            {182,42},
            {188,42},
            {185,48}};
winset(control_win);
char s[32];
pushmatrix();
ortho2(0.0, 769.0, 0.0, 100.0);
RGBcolor(255,255,255);
clear();

//Go & Reset Buttons
RGBcolor(200,200,200);
rectf(475.0, 25.0, 525.0, 75.0);
rectf(550.0, 25.0, 605.0, 75.0);
rectf(625.0, 25.0, 675.0, 75.0);
rectf(700.0, 25.0, 750.0, 75.0);
```

```
RGBcolor(0,0,0);
cmov2(487, 50);
charstr("Body");
cmov2(482, 40);
charstr("Moment");
cmov2(551, 50);
charstr("Inertial");
cmov2(558, 40);
charstr("Moment");
cmov2(639,50);
charstr("Spin");
cmov2(643,40);
charstr("up");
cmov2(710,45);
charstr("Reset");

// Angular Velocity
RGBcolor(0,0,255);
rectf(150.0, 40.0, 180.0, 70.0);
RGBcolor(200,200,200);
rectf(140.0, 40.0, 150.0, 70.0);
rectf(180.0, 40.0, 190.0, 70.0);
// Draw up and down arrows
RGBcolor(0,0,0);
polf2(3,pt);
polf2(3,pt2);
pt[0][1] = pt[0][1] + 10.0;
pt[1][1] = pt[1][1] + 10.0;
pt[2][1] = pt[2][1] + 10.0;
pt2[0][1] = pt2[0][1] + 10.0;
pt2[1][1] = pt2[1][1] + 10.0;
pt2[2][1] = pt2[2][1] + 10.0;
polf2(3,pt);
polf2(3,pt2);
pt[0][1] = pt[0][1] + 10.0;
pt[1][1] = pt[1][1] + 10.0;
pt[2][1] = pt[2][1] + 10.0;
pt2[0][1] = pt2[0][1] + 10.0;
pt2[1][1] = pt2[1][1] + 10.0;
pt2[2][1] = pt2[2][1] + 10.0;
polf2(3,pt);
polf2(3,pt2);
RGBcolor(255,255,0);
cmov2(155, 61);
sprintf(s, "%.0f", (double) x);
charstr(s);
cmov2(155, 51);
sprintf(s, "%.0f", (double) y);
charstr(s);
cmov2(155, 41);
sprintf(s, "%.0f", (double) z);
charstr(s);

// External Moment
```

```
RGBcolor(0,0,255);
rectf(220.0, 40.0, 250.0, 70.0);
RGBcolor(200,200,200);
rectf(210.0, 40.0, 220.0, 70.0);
rectf(250.0, 40.0, 260.0, 70.0);
// Draw up and down arrows
RGBcolor(0,0,0);
pt[0][0] = pt[0][0] + 70.0;
pt[1][0] = pt[1][0] + 70.0;
pt[2][0] = pt[2][0] + 70.0;
pt2[0][0] = pt2[0][0] + 70.0;
pt2[1][0] = pt2[1][0] + 70.0;
pt2[2][0] = pt2[2][0] + 70.0;
polf2(3,pt);
polf2(3,pt2);
pt[0][1] = pt[0][1] - 10.0;
pt[1][1] = pt[1][1] - 10.0;
pt[2][1] = pt[2][1] - 10.0;
pt2[0][1] = pt2[0][1] - 10.0;
pt2[1][1] = pt2[1][1] - 10.0;
pt2[2][1] = pt2[2][1] - 10.0;
polf2(3,pt);
polf2(3,pt2);
pt[0][1] = pt[0][1] - 10.0;
pt[1][1] = pt[1][1] - 10.0;
pt[2][1] = pt[2][1] - 10.0;
pt2[0][1] = pt2[0][1] - 10.0;
pt2[1][1] = pt2[1][1] - 10.0;
pt2[2][1] = pt2[2][1] - 10.0;
polf2(3,pt);
polf2(3,pt2);
RGBcolor(255,255,0);
cmov2(225, 61);
sprintf(s, "%.0f", (double) t1);
charstr(s);
cmov2(225, 51);
sprintf(s, "%.0f", (double) t2);
charstr(s);
cmov2(225, 41);
sprintf(s, "%.0f", (double) t3);
charstr(s);

// Duration & Magnitude
RGBcolor(0,0,255);
rectf(290.0, 60.0, 330.0, 70.0);
rectf(290.0, 40.0, 330.0, 50.0);
RGBcolor(200,200,200);
rectf(280.0, 60.0, 290.0, 70.0);
rectf(330.0, 60.0, 340.0, 70.0);
rectf(280.0, 40.0, 290.0, 50.0);
rectf(330.0, 40.0, 340.0, 50.0);
// Draw up and down arrows
RGBcolor(0,0,0);
pt[0][0] = pt[0][0] + 70.0;
```

```
pt[1][0] = pt[1][0] + 70.0;
pt[2][0] = pt[2][0] + 70.0;
pt2[0][0] = pt2[0][0] + 80.0;
pt2[1][0] = pt2[1][0] + 80.0;
pt2[2][0] = pt2[2][0] + 80.0;
polf2(3,pt);
polf2(3,pt2);
pt[0][1] = pt[0][1] + 20.0;
pt[1][1] = pt[1][1] + 20.0;
pt[2][1] = pt[2][1] + 20.0;
pt2[0][1] = pt2[0][1] + 20.0;
pt2[1][1] = pt2[1][1] + 20.0;
pt2[2][1] = pt2[2][1] + 20.0;
polf2(3,pt);
polf2(3,pt2);
RGBcolor(255,255,0);
cmov2(295, 61);
sprintf(s, "%.1f", duration);
charstr(s);
cmov2(295, 41);
sprintf(s, "%.0e", mag);
charstr(s);

//Clocks
RGBcolor(255,0,0);
rectf(355.0, 10.0, 460.0, 80.0);
RGBcolor(0,0,0);
cmov2(365.0,71.0);
charstr("Session Time");
cmov2(360.0,41.0);
charstr("Moment Applied");
RGBcolor(255,255,255);

cmov2(380, 60);
sprintf(s, "%.1f", total);
charstr(s);
cmov2(380, 30);
sprintf(s, "%.2f", elapsed);
charstr(s);

//window text
RGBcolor(0,0,0);
cmov2(10.0,90.0);
charstr("Gyroscopic Stiffness Control Window");
cmov2(10.0,75.0);
charstr("Shape");
cmov2(92.0,75.0);
charstr("Size");
cmov2(140.0,75.0);
charstr("Ang Vel");
cmov2(197.0,61.5);
charstr("X");
cmov2(197.0,51.5);
charstr("Y");
```

```
cmov2(197.0,41.5);
charstr("Z");
cmov2(215.0,75.0);
charstr("Moment");
cmov2(280.0,71.0);
charstr("Duration");
cmov2(280.0,51.0);
charstr("Magnitude");

// Select between the 3 base objects
RGBcolor(0,0,255);
rectf(10.0, 40.0, 70.0, 70.0);
RGBcolor(255,255,0);
cmov2(15.0,61.5);
charstr("Sphere");
cmov2(15.0,51.5);
charstr("Block");
cmov2(15.0,41.5);
charstr("Cylinder");

switch(object)
{
    case 1:
    rectf(10.0, 60.0, 70.0, 70.0);
    RGBcolor(0,0,255);
    cmov2(15.0,61.5);
    charstr("Sphere");
    break;

    case 2:

    rectf(10.0, 50.0, 70.0, 60.0);
    RGBcolor(0,0,255);
    cmov2(15.0,51.5);
    charstr("Block");
    break;

    case 3:
    rectf(10.0, 40.0, 70.0, 50.0);
    RGBcolor(0,0,255);
    cmov2(15.0,41.5);
    charstr("Cylinder");
    break;

    default:
    rectf(10.0, 60.0, 70.0, 70.0);
    RGBcolor(0,0,255);
    cmov2(15.0,61.5);
    charstr("Sphere");
    break;

}

//Object Size
```

```
RGBcolor(0,0,255);
rectf(90.0, 40.0, 120.0, 70.0);
RGBcolor(200,200,200);
rectf(80.0, 40.0, 90.0, 70.0);
rectf(120.0, 40.0, 130.0, 70.0);
// Draw up and down arrows
RGBcolor(0,0,0);
pt[0][0] = pt[0][0] - 200.0;
pt[1][0] = pt[1][0] - 200.0;
pt[2][0] = pt[2][0] - 200.0;
pt2[0][0] = pt2[0][0] - 210.0;
pt2[1][0] = pt2[1][0] - 210.0;
pt2[2][0] = pt2[2][0] - 210.0;
polf2(3,pt);
polf2(3,pt2);
pt[0][1] = pt[0][1] - 10.0;
pt[1][1] = pt[1][1] - 10.0;
pt[2][1] = pt[2][1] - 10.0;
pt2[0][1] = pt2[0][1] - 10.0;
pt2[1][1] = pt2[1][1] - 10.0;
pt2[2][1] = pt2[2][1] - 10.0;
polf2(3,pt);
polf2(3,pt2);
pt[0][1] = pt[0][1] - 10.0;
pt[1][1] = pt[1][1] - 10.0;
pt[2][1] = pt[2][1] - 10.0;
pt2[0][1] = pt2[0][1] - 10.0;
pt2[1][1] = pt2[1][1] - 10.0;
pt2[2][1] = pt2[2][1] - 10.0;
polf2(3,pt);
polf2(3,pt2);
RGBcolor(255,255,0);
cmov2(91, 61);
sprintf(s, "%.1f", size[0]);
charstr(s);
cmov2(91, 51);
sprintf(s, "%.1f", size[1]);
charstr(s);
cmov2(91, 41);
sprintf(s, "%.1f", size[2]);
charstr(s);
popmatrix();
swapbuffers();
}

void stat_controls(double x, double y, double z, double am, double mass, double Ix, double Iy, double Iz,
vector3D* old_av)
{
int i;
winset(main_win);
char s[32];

RGBcolor(255,0,0);
cmovs(10, 15, -30);
```

```
charstr("X velocity");
cmovs(15, 15, -30);
sprintf(s, "%.3f", x);
charstr(s);
cmovs(-20, 14, -30);
charstr("Ixx");
cmovs(-18, 14, -30);
sprintf(s, "%.1f", Ix);
charstr(s);

RGBcolor(0,0,255);
cmovs(10, 14, -30);
charstr("Y velocity");
cmovs(15, 14, -30);
sprintf(s, "%.3f", y);
charstr(s);
cmovs(-20, 13, -30);
charstr("Iyy");
cmovs(-18, 13, -30);
sprintf(s, "%.1f", Iy);
charstr(s);

RGBcolor(0,0,0);
cmovs(10, 13, -30);
charstr("Z velocity");
cmovs(15, 13, -30);
sprintf(s, "%.3f", z);
charstr(s);
cmovs(-20, 12, -30);
charstr("Izz");
cmovs(-18, 12, -30);
sprintf(s, "%.1f", Iz);
charstr(s);

cmovs(10, 12, -30);
charstr("Angular Momentum");
cmovs(17, 12, -30);
sprintf(s, "%.1f", am);
charstr(s);

cmovs(-20, 15, -30);
charstr("Mass");
cmovs(-18, 15, -30);
sprintf(s, "%.1f", mass);
charstr(s);

RGBcolor(200,200,200);
rectf(-3.2, -4.1, 3.25, -1.8);
RGBcolor(100,100,100);
rectf(-2.8, -3.01, 3.25, -2.99);

RGBcolor(0,0,0);
cmovs(-3, -3, 0);
charstr("0");
```

```
cmovs(-3, -2, 0);
charstr("10");
cmovs(-3, -4, 0);
charstr("-10");

for(i=0;i<300;i++) //draw x graph
{
    RGBcolor(255,0,0);
    if((old_av[i])[0] < 10.0 && (old_av[i])[0] > -10.0)
        rectf(-2.75 + 0.02 * i,-2.99 + .1 * (old_av[i])[0],-2.73 + 0.02 * i,-3.01 + .1 * (old_av[i])[0]);
    else
        if((old_av[i])[0] < 0.0)
            rectf(-2.75 + 0.02 * i,-3.99,-2.73 + 0.02 * i,-4.01);
        else
            rectf(-2.75 + 0.02 * i,-1.99,-2.73 + 0.02 * i,-2.01);

    RGBcolor(0,0,255);  //draw y graph
    if((old_av[i])[1] < 10.0 && (old_av[i])[1] > -10.0)
        rectf(-2.75 + 0.02 * i,-2.99 + .1 * (old_av[i])[1],-2.73 + 0.02 * i,-3.01 + .1 * (old_av[i])[1]);
    else
        if((old_av[i])[1] < 0.0)
            rectf(-2.75 + 0.02 * i,-3.99,-2.73 + 0.02 * i,-4.01);
        else
            rectf(-2.75 + 0.02 * i,-1.99,-2.73 + 0.02 * i,-2.01);

    RGBcolor(0,0,0);  //draw z graph
    if((old_av[i])[2] < 10.0 && (old_av[i])[2] > -10.0)
        rectf(-2.75 + 0.02 * i,-2.99 + .1 * (old_av[i])[2],-2.73 + 0.02 * i,-3.01 + .1 * (old_av[i])[2]);
    else
        if((old_av[i])[2] < 0.0)
            rectf(-2.75 + 0.02 * i,-3.99,-2.73 + 0.02 * i,-4.01);
        else
            rectf(-2.75 + 0.02 * i,-1.99,-2.73 + 0.02 * i,-2.01);

}
}

void attach_eye_to(vector3D* v, int* i)
{
    if (eye_display_field != NULL)
    {
        *eye_display_field = 1;
    }
    eye = v;
    eye_display_field = i;
    *eye_display_field = 0;
    twist = 0;
}

void attach_target_to(vector3D* v)
{
    target = v;
    twist = 0;
}
```

```
void set_eye_to(double x, double y, double z)
{
    if (eye_display_field != NULL)
    {
        *eye_display_field = 1;
        eye_display_field = NULL;
    }
    eye = new vector3D(x, y, z);
    twist = 0;
}

void set_target_to(double x, double y, double z)
{
    target = new vector3D(x, y, z);
    twist = 0;
}

void view()
{
    swapbuffers();
    czclear(0xFFFF7200,getgdesc(GD_ZMAX));
    loadmatrix(un);
    perspective(450,1.25,0.2,10000.0);
    lookat((*eye)[0],(*eye)[1],(*eye)[2],(*target)[0],(*target)[1],(*target)[2], (int) (twist *
572.957795131));
    display_this_object(lightobj);

}

void view(quaternion q, vector3D new_eye, int view_axis)
{
    Matrix rt = { 1.0, 0.0, 0.0, 0.0,
        0.0, 1.0, 0.0, 0.0,
        0.0, 0.0, 1.0, 0.0,
        0.0, 0.0, 0.0, 1.0};
    matrix3x3 rotation, axis;
    swapbuffers();
    czclear(0xFFFF7200,getgdesc(GD_ZMAX));
    loadmatrix(un);
    perspective(450,1.25,0.2,10000.0);
    switch(view_axis)
    {
        //Negative Y axis
        case -2:
        axis.DCM_x_rotation(1.5707963268);
        break;

        //Negative X axis
        case -1:
        axis.DCM_y_rotation(-1.5707963268);
        break;
```

```
            //Positive X axis
            case 1:
            axis.DCM_y_rotation(1.5707963268);
            break;

            //Positive Y axis
            case 2:
            axis.DCM_x_rotation(-1.5707963268);
            break;

            //Positive Z axis
            case 3:
            axis.DCM_y_rotation(3.14159265359);
            break;

            default:
            break;
        }
        rotation.DCM_body_to_world(q);
        rotation = rotation * axis;
        new_eye = new_eye * -1;
        // load rotation elements
        rt[0][0] = rotation[0];
        rt[1][0] = rotation[3];
        rt[2][0] = rotation[6];
        rt[3][0] = 0.0;
        rt[0][1] = rotation[1];
        rt[1][1] = rotation[4];
        rt[2][1] = rotation[7];
        rt[3][1] = 0.0;
        rt[0][2] = rotation[2];
        rt[1][2] = rotation[5];
        rt[2][2] = rotation[8];
        rt[3][2] = 0.0;
        rt[3][0] = 0;
        rt[3][1] = 0;
        rt[3][2] = 0;
        rt[3][3] = 1.0;
        multmatrix(rt);  // rotate view
    // load translational matrix
        rt[0][0] = 1;
        rt[1][0] = 0;
        rt[2][0] = 0;
        rt[3][0] = 0;
        rt[0][1] = 0;
        rt[1][1] = 1;
        rt[2][1] = 0;
        rt[3][1] = 0;
        rt[0][2] = 0;
        rt[1][2] = 0;
        rt[2][2] = 1;
        rt[3][2] = 0;
        rt[3][0] = new_eye[0];
        rt[3][1] = new_eye[1];
```

```
    rt[3][2] = new_eye[2];
    rt[3][3] = 1.0;
    multmatrix(rt);  // translate eye point
    display_this_object(lightobj);
}
void view_axis()
{
    display_this_object(axis);
}

void rotate_view(int angle)
{
    twist = angle;
}
#endif
```

APPENDIX J

Menu Code

A. HEADER FILE

```
#ifndef MENU_H
#define MENU_H
#include "menu.H"
#include <gl.h>
#include <device.h>
#include <stdio.h>

initialize_menu();
queue_test();

#endif
```

B. SOURCE FILE

```
#ifndef MENU_C
#define MENU_C
#include "menu.H"
#include "time.H"
#include <iostream.h>

long mainmenu;
long hititem;
short value;
double wx, wy;

long makethemenus()
/* this routine performs all the menu construction calls */
{
long topmenu;

  topmenu = defpup("Dynamics Visualizer %t | Exit %x99");
  return(topmenu);
}

void initialize_menu()
{
  mainmenu = makethemenus();  //GL function
}

void processmenuhit(long hititem)
{
  switch(hititem)
  {
      case 99:
        exit(0);
        break;

      default:
        break;
  } /* end switch */
}

int queue_test()
{
    hititem = 0;
    while(qtest())
    {
        switch(qread(&value))
        {
            case MENUBUTTON:
                if(value == 1)
                {
                    mmode(MSINGLE);
                    hititem = dopup(mainmenu);
                    mmode(MVIEWING);
                    processmenuhit(hititem);
```

```
            reset_time();
            }
        break;

    case LEFTMOUSE:
        wx = getvaluator(MOUSEX);
        wy = getvaluator(MOUSEY);
        hititem = (wx * 100000) + wy;  //encode mouse location
        break;

    case REDRAW:
        reshapeviewport();
        break;

    case UPARROWKEY:
        hititem = 100;
        break;

    case DOWNARROWKEY:
        hititem = 101;
        break;

    case LEFTARROWKEY:
        hititem = 102;
        break;

    case RIGHTARROWKEY:
        hititem = 103;
        break;

    case EQUALKEY:
        hititem = 104;
        break;

    case MINUSKEY:
        hititem = 105;
        break;

    case SPACEKEY:
        hititem = 106;
        break;

    case F1KEY:
        hititem = 111;
        break;

    case F2KEY:
        hititem = 112;
        break;

    case F3KEY:
        hititem = 113;
        break;
```

```
            case F4KEY:
                hititem = 114;
                break;

            case F5KEY:
                hititem = 115;
                break;

            case F6KEY:
                hititem = 116;
                break;

            case F7KEY:
                hititem = 117;
                break;

            case F8KEY:
                hititem = 118;
                break;

            case F9KEY:
                hititem = 119;
                break;

            case F10KEY:
                hititem = 120;
                break;

            case F11KEY:
                hititem = 121;
                break;

            case F12KEY:
                hititem = 122;
                break;

            default:
                break;
        }
    }
    return (int) hititem;
}
#endif
```

INITIAL DISTRIBUTION LIST

1.	Superintendent Attn: Library, Code 1424 Naval Postgraduate School Monterey, California 93940-5000	2
2.	Department of Aeronautics & Astronautics Attn: Professor I Michael Ross, Code AA/RO Naval Postgraduate School Monterey, California 93940-5000	7
3.	Department of Computer Science Attn: Professor Robert B. McGhee, Code CS/MZ Naval Postgraduate School Monterey, California 93940-5000	2
4.	Keith L. Haynes 3201 Lowson Blvd Delray Beach, FL 33445	2
5.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2