Calhoun: The NPS Institutional Archive

DSpace Repository

Theses and Dissertations | 1. Thesis and Dissertation Collection, all items

1994-03

Graphical simulation of articulated rigid body system kinematics with collision detection

Goetz, John Robert

Monterey, California. Naval Postgraduate School

https://hdl.handle.net/10945/30911

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

Downloaded from NPS Archive: Calhoun

http://www.nps.edu/library

Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

Graphical Simulation of Articulated Rigid Body System Kinematics with Collision Detection

by

John Robert Goetz

23 March 1994

Thesis Advisor: Robert B. McGhee
Thesis Co-Advisor: Michael J. Zyda

Approved for public release; distribution is unlimited.

Thesis
G5282

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5101

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1994	Master's Thesis

4. TITLE AND SUBTITLE
Graphical Simulation of Articulated Rigid Body System Kinematics with Collision Detection

5. FUNDING NUMBERS

6. AUTHOR(S)
Goetz, John Robert

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 United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE
A

13. ABSTRACT *(Maximum 200 words)*

The area of concern addressed by this thesis is the development of a 3-D visual simulation to aid in the testing of ground detection and foot placement algorithms for an articulated walking robot's foot pads on uneven terrain.

Several collision detection algorithms and terrain mapping techniques were studied to determine which approach would lend itself readily to the rapid detection of initial ground contact and the required orientation needed to place each foot firmly on the ground.

As a result of these studies, a real-time, realistic and aesthetically pleasing graphical simulation for the testing of control software for articulated walking machines has been developed which utilizes the Silicon Graphics 3-D visual simulation toolkit, *Performer*. Not only is rapid ground contact sensing and foot orientation possible, it is accomplished without using extraneous data structures making the algorithm generic enough to use on any terrain model.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Visual Simulation, Robotics, Walking Machines, Aquarobot, Performer	83
	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	Unlimited

Approved for public release; distribution is unlimited

Graphical Simulation of Articulated Rigid Body System Kinematics with Collision Detection

by
John Robert Goetz
Lieutenant USN
B. S., Maine Maritime Academy, 1987

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

March 1994

Author: [redacted]

John Robert Goetz

Approved By: [redacted]

Robert B. McGhee, Thesis Advisor

[redacted]

Michael J. Zyda, Thesis Co-Advisor

[redacted]

Mr. Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

The area of concern addressed by this thesis is the development of a 3-D visual simulation to aid in the testing of ground detection and foot placement algorithms for an articulated walking robot's foot pads on uneven terrain.

Several collision detection algorithms and terrain mapping techniques were studied to determine which approach would lend itself readily to the rapid detection of initial ground contact and the required orientation needed to place each foot firmly on the ground.

As a result of these studies, a real-time, realistic and aesthetically pleasing graphical simulation for the testing of control software for articulated walking machines has been developed which utilizes the Silicon Graphics 3-D visual simulation toolkit, *Performer*. Not only is rapid ground contact sensing and foot orientation possible, it is accomplished without using extraneous data structures making the algorithm generic enough to use on any terrain model.

Thesis
G5282
c.2

ACKNOWLEDGEMENTS

I would like to thank my thesis advisors Dr. McGhee and Dr. Zyda for there time, patients, understanding and coping with my management by crisis approach to thesis writing. I thank Dave Pratt (Dr. Dave) for being there at 0300 to answer questions and shed light on areas I should have paid more attention to while he was teaching class. I would also like to thank the other members of the Aquarobot project (past and present) for their shared knowledge and assistance. And most of all, I give my undying appreciation to my wife Lisa and our two sons Anthony and Craig. I couldn't always be there for them, but they were always there for me. Thank You.

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5101

I. INTRODUCTION 1
A. AQUAROBOT 1
B. NPS INVOLVEMENT 2
C. SUMMARY OF CHAPTERS 3
II. SURVEY OF PREVIOUS WORK 4
A. WALKING MACHINES 4
B. GRAPHICAL SIMULATION 4
C. PRIOR CONTRIBUTIONS TO THE AQUAROBOT PROJECT 4
1. Control Software 4
2. Simulation 5
III. GRAPHICAL SIMULATION USING PERFORMER 6
A. HIERARCHICAL DATABASE STRUCTURE 6
1. pfScene 6
2. pfGroup 7
3. pfGeode 7
B. IMPLEMENTATION 7
1. pfSCS 8
2. pfDCS 8
3. pfGeoSet 8
4. pfGeoState 8
5. pfSeg 9
6. pfIsect 9
7. pfPlane 9
8. pfCylinder 10
C. COORDINATE SYSTEM 11
1. Joint Axis Notation 11
D. AQUAROBOT CONSTRUCTION 12
1. Performer Node Structure 13
a. Articulations 14
b. Torso Local Coordinate System 15
c. Foot Pad Local Coordinate System 15
2. Joint Limits 16
a. Ball Joints 16
IV. ENVIRONMENT MODELING FOR FOOT PLACEMENT 17
A. TERRAIN MODEL 17
1. Characteristics 17
B. LIMITATIONS DUE TO GAIT ALGORITHM 18
C. TERRAIN IDENTIFICATION 19
D. UNDERWATER ENVIRONMENT 20
E. RESULTS 20
V. GROUND CONTACT DETERMINATION 21
A. BOUNDING VOLUMES 21
B. INTERSECTING SEGMENTS 22

1. A Single Center Segment 23
a. Results 23
2. Multiple Segments 24
a. Foot Pad Divisions 25
b. Creating an Artificial Plane 25
c. Correcting the Artificial Plane 26
d. Results 26
3. Speeding Up the Process 28
VI. RESULTS 29
A. SIMULATION DESCRIPTION 29
1. Keyboard Responses 29
VII. CONCLUSIONS AND RECOMMENDATIONS 30
A. RECOMMENDATIONS 30
B. FUTURE WORK 30
APPENDIX A: SIMULATION SOURCE CODE 31
APPENDIX B: SIMULATION SUPPORT CODE 56
APPENDIX C: LOADING NPSGDL2 FILES INTO PERFORMER 65
1. PURPOSE 65
2. GENERAL DESCRIPTION 65
3. PRIMITIVES 66
4. COORDINATE SYSTEM 67
5. MATRIX TRANSFORMATIONS 67
a. LOADING A MATRIX 68
6. RENDERING ATTRIBUTES 68
a. TEXTURE COORDINATE GENERATION 69
7. COMMENTS 69
8. FUTURE WORK 69
9. TESTING 69
10. SOURCE CODE AND LIBRARY 69
LIST OF REFERENCES 71
INITIAL DISTRIBUTION LIST 73

LIST OF FIGURES

Figure 1: Aquarobot Being Lowered into Yokosuka Bay 1
Figure 2: Graphical Model of the Aquarobot in a Simulated Underwater Environment ..2
Figure 3: Simple Performer Hierarchy 7
Figure 4: A pfCylinder around pfSegs 10
Figure 5: Robotics Joint Axis Notation 11
Figure 6: Graphical Representation of Aquarobot 12
Figure 7: Performer Node Structure for Aquarobot 13
Figure 8: Coordinate Systems for Torso and a Typical Leg With Foot Pad 14
Figure 9: Local Coordinate Systems After Being Rotated 15
Figure 10: Motion of Foot When Joint Limit is Reached 16
Figure 11: Different Terrain Types Used in Terrain Model 18
Figure 12: Aquarobot Standing in Water 19
Figure 13: Bounding Box for an Upper Leg 21
Figure 14: Growth of a Bounding Volume 22
Figure 15: Foot Placement With a Single Segment at the Center of the Foot 24
Figure 16: Result of Using a Single Segment From the Center of the Foot 24
Figure 17: Placement of Segments 25
Figure 18: Foot Orientation Using an Artificial Plane 26
Figure 19: Final Foot Placement 27
Figure 20: Result of Having all Three Ground Contact Points on One Side of the Foot .27
Figure C1: Converting a polygon to a trimesh 66
Figure C2: GDL Loader Matrix Array 67

I. INTRODUCTION

A. AQUAROBOT

Aquarobot (Figure 1) is a 6 legged articulated walking robot developed by the Port and Harbour Research Institute of Japan (PHRI), for the purpose of surveying the construction of an undersea rock mound seawall foundation. This seawall is being built to protect

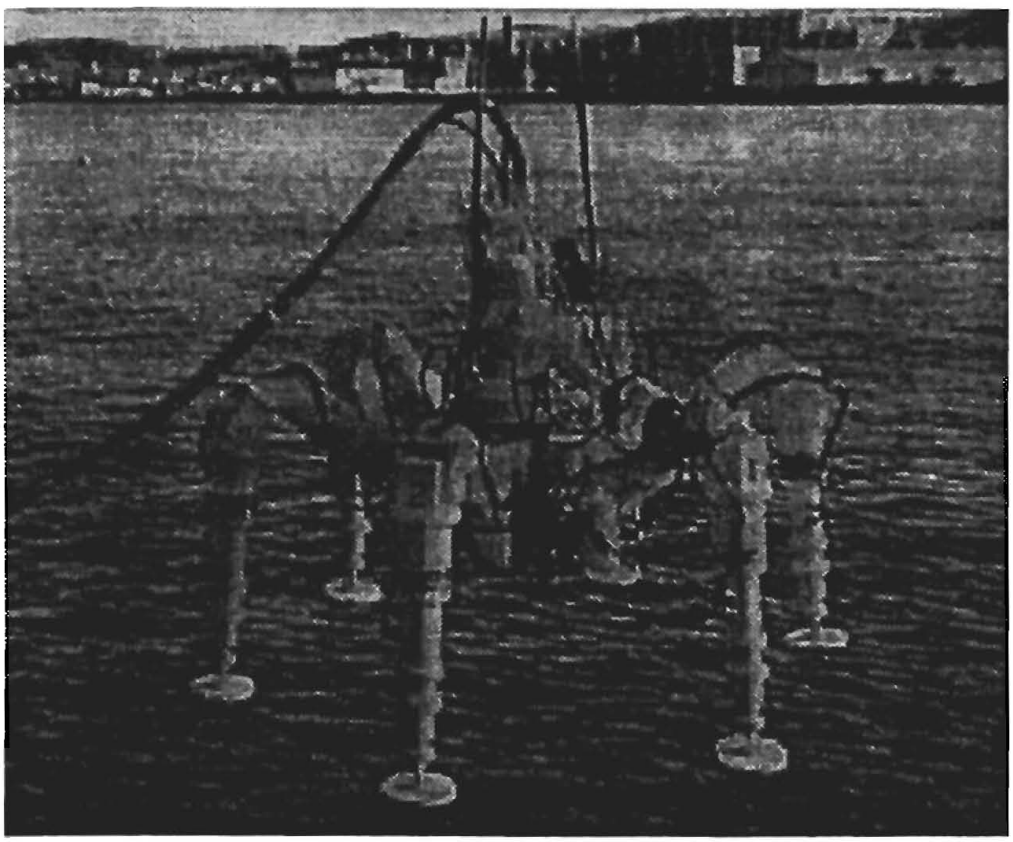


Figure 1: Aquarobot Being Lowered into Yokosuka Bay

portions of Japan's coastline from tidal waves (tsunamis), which have inflicted severe damage and loss of life in the past. The primary functions of *Aquarobot* are to: 1) measure

the flatness of the rock mound which will support a caisson barrier, and 2) survey the placement of the cement caissons utilizing its on-board camera.

After completing several successful inspections, it was determined that human divers could perform the surveys more cost effectively than *Aquarobot* due to the number of people needed to lower, control, and recover *Aquarobot* using the current vehicle software and "man in the loop" operational concept [Ref. 1].

B. NPS INVOLVEMENT

The Naval Postgraduate School, under a grant from the National Science Foundation (NSF), is developing gait algorithms to increase the speed and improve the efficiency the *Aquarobot*. A long range goal of this project is to achieve fully autonomous operation of *Aquarobot* and similar vehicles [Ref. 2, 3].

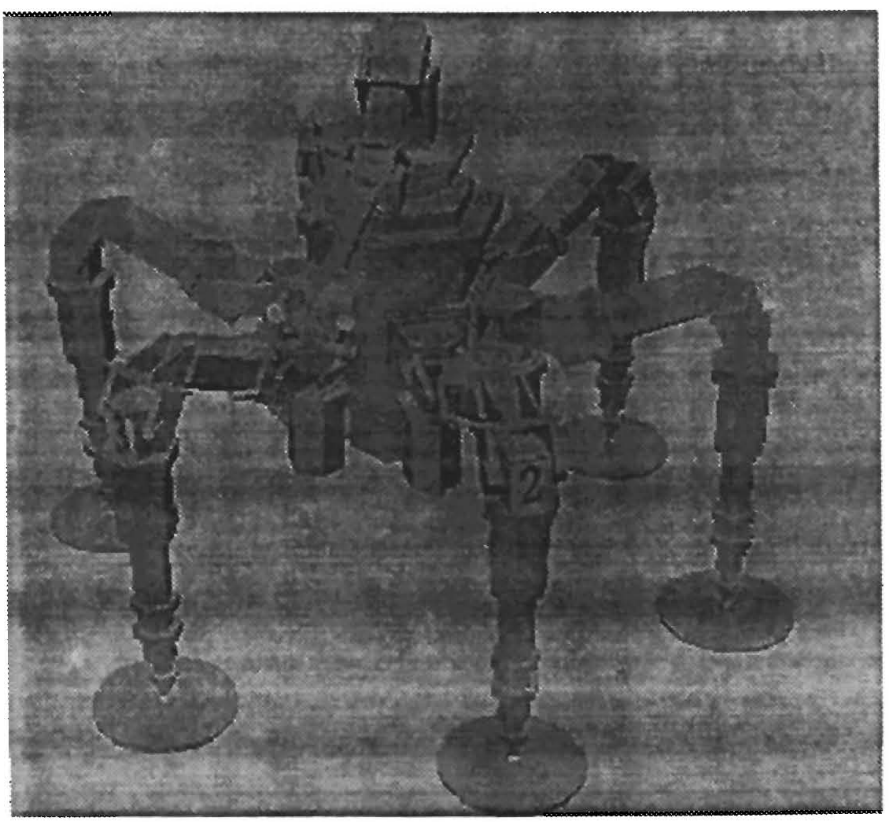


Figure 2: Graphical Model of the *Aquarobot* in a Simulated Underwater Environment

To ensure that new software is performing correctly prior to actual use, a graphical simulation is being developed to test all foreseeable situations. Once the software has performed adequately in a controlled graphical environment, then it will be tested on the actual *Aquarobot.*

Figure 2 depicts the latest 3D model of *Aquarobot* in a simulated underwater environment. Even though this is the correct environment for the robot, the simulated murky water makes the robot more difficult to see and slows down the simulation. For these reasons, the simulation results shown in the remainder of this thesis have the appearance of being performed on dry land, when in actuality the robot behaves as if it were in an underwater environment.

C. SUMMARY OF CHAPTERS

Chapter II is a brief history of the advances computer simulation has taken in becoming a tool for project development. Also included is a look at where walking robots stand today and a survey of work accomplished by individuals previously involved with the *Aquarobot* project at the Naval Postgraduate School. Chapter III discuses the software tool *Performer* which is used to enhance the graphical simulation and provides a description of the graphical model. Chapter IV describes environment modeling and terrain identification. Chapter V deals with the algorithms used to detect foot contact with the ground and with the orientation of *Aquarobot's* foot pads to conform to uneven terrain. Chapter VI is a complete description of the simulation and its use. Chapter VII presents the conclusions of this research and recommendations for follow on work. Appendices A and B contain program code. Appendix C provides a description of a NPSGDL to Performer loader. Source code for this loader can be obtained by contacting the author.

II. SURVEY OF PREVIOUS WORK

A. WALKING MACHINES

Researchers began seriously investigating the use of terrain-adaptive vehicles around 1970 for the purposes of off-road transportation, space assembly and forestry [Ref. 4]

The majority of walking machines today are research prototypes limited to navigating smooth, non-compliant surfaces. [Ref. 5] discusses the first legged vehicle model to incorporate foot slippage and sinkage and foot placement on rocks, uneven surfaces and slopes.

Aquarobot is the first walking machine constructed for a specific purpose, the inspection of an undersea rock wall foundation off Japan's coastline [Ref. 6].

B. GRAPHICAL SIMULATION

Using computer simulations to aid in the testing of products is by no means emerging technology. But the idea of creating a virtual environment for the testing of a product before the product is even built is relatively new [Ref. 7].

C. PRIOR CONTRIBUTIONS TO THE AQUAROBOT PROJECT

The Naval Postgraduate school first got involved in the Aquarobot project in 1992. Since then it has been the subject of four masters theses (including this one), one Ph.D dissertation and numerous papers. Current research is divided into two distinct categories, control software and simulation.

1. Control Software

New control software is being developed which will increase the speed and efficiency of Aquarobot's motion (gait). The gait currently used is a discrete tripod gait. Three legs are moved in a group and the torso only moves when all six feet are on the ground. The new gait algorithm (wave gait) allows for the control of individual legs to

maintain a "smooth and dynamic trajectory" of the torso [Ref. 8], and takes optimal advantage of the range of motion for the legs while maintaining a sufficient margin for stability [Ref. 1].

2. Simulation

[Ref. 9] establishes a kinematics model based on the Modified Danevit-Hartenberg (Craig) method of representing articulated joints. This model was then used in the development of the simplified dynamic simulation which models the DC motors that control the motion of the joints [Ref. 10]. A complete hydrodynamics model is being developed [Ref. 11] which will provide the base line data to verify the dynamic simulation. This is necessary since it is unluckily that the hydrodynamic simulation will run in real-time.

The work presented in this thesis develops a 3D simulation to display the results of the dynamic simulation. It also deals with the issue of detecting foot collisions with the ground and foot pad orientation on uneven terrain.

III. GRAPHICAL SIMULATION USING PERFORMER

IRIS *Performer* is a 3D software toolkit developed by Silicon Graphics Inc. (SGI) for developing real-time visual simulations on their graphics systems. Performer combines an application programming interface (API) with a high-performance rendering library to take full advantage of the advanced hardware improvements of SGI's Reality Engine visual simulation system[Ref. 12].

There is no provision for off-line storage of this visual database, so a conversion program must be used to create the run-time structure from some other database format. The current implementation of the Aquarobot Simulator uses a locally developed visual database called *NPS Graphical Description Language* (NPSGDL2) to describe the articulated pieces which the robot is comprised of. A loader then reads the NPSGDL2 descriptions, converts them to Performer format, and stores them in memory for later rendering. The NPSGDL2 loader is discussed in detail in Appendix C. The graphical model is further discussed in Section D.

A. HIERARCHICAL DATABASE STRUCTURE

Performer uses a spatially organized hierarchical database structure that is created at run-time for scene management. A hierarchical visual database is a tree-like structure which contains all of the transformations and geometry needed to render a 3D graphics image. A node may have more than one parent which prevents the database from being a true tree. Figure 3 describes a simplistic Performer database.

1. pfScene[1]

A pfScene is the root node of the visual database It functions in the same manner as a pfGroup with the exception that it can not be a child of any other pfGroup.

1. All Performer types and functions begin with 'pf' to indicate it is a Performer feature.

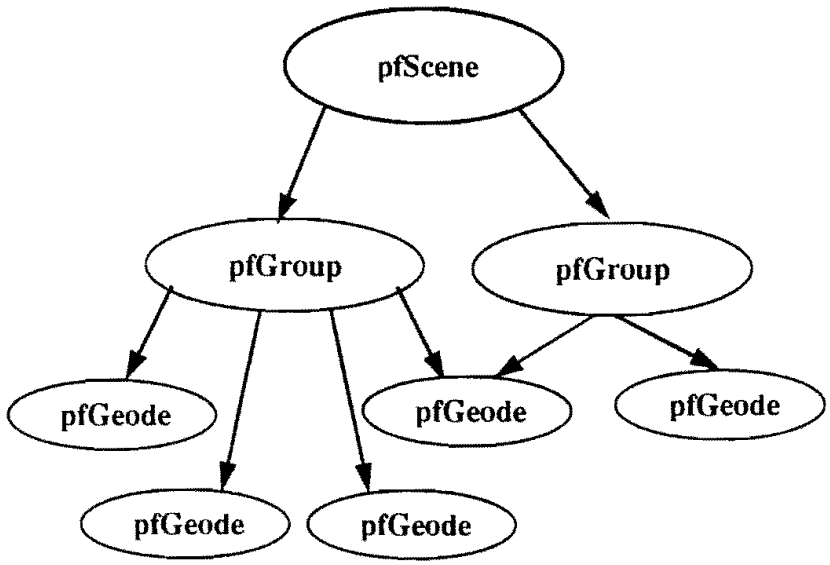


Figure 3: Simple Performer Hierarchy

2. pfGroup

A pfGroup is a branch node in the visual database and can have any other type of node as a child except for the pfScene node. It is used to group other nodes in the database spatially. A pfGroup may share information with another pfGroup. This allows multiple instances of a model at different locations and orientations while only having a single model description in memory at run time.

3. pfGeode

A pfGeode (Geometry Node) is a leaf node that contains the graphics information which defines the visual simulation. A pfGeode is comprised of pfGeoSets and pfGeoStates which are described below. A pfGeode may have more than one parent.

B. IMPLEMENTATION

To have a better understanding of the graphical model of *Aquarobot* and of the foot placement algorithm, it is beneficial to comprehend a few of the types, structures and functions that Performer provides. For a more complete description of Performer see [Ref. 12, 13].

1. pfSCS

A pfSCS is a Static Coordinate System. Position, orientation, and scale transformations can be applied when a node is added to the visual database. Since this is a static coordinate system, these transformations are only applied once. This allows instancing of a single model at several locations within the virtual world or allows for the placement of articulated parts so that movement of these parts can simply be done with a rotation about the joint axis. A pfSCS is a type of pfGroup.

2. pfDCS

A pfDCS is a Dynamic Coordinate System used to translate and rotate objects at run-time. These nodes are used for all object movement, from articulated parts, to actually moving objects around in a virtual world. A pfDCS is a type of pfGroup.

3. pfGeoSet

A pfGeoSet ("Geometry Set") is a collection of like geometry[2]. The elements within a pfGeoSet are called primitives and may be any one of points, lines, linestrips, tris (3 sided polygons), quads (4 sided polygons), or tristrips[3]. Primitives are described using vertex lists which may be indexed or not indexed depending on whether or not memory needs to be conserved.

4. pfGeoState

A pfGeoState describes the graphics state for the pfGeoSet for which it is applied to. The state describes such features as material, textures, texture environment and transparency.

---

2. Like geometry refers to a collection of primitives which are of the same type, and have the same state information.
3. A tristrip is a collection of three sided polygons specified by a sequence of the three most recent vertices.

5. pfSeg

A pfSeg is a non-graphical line segment described by a starting position, a direction and a length. These line segments, or intersecting rays, are used for intersection testing with the stored geometry.

6. pfIsect

A pfIsect is a structure which contains information about the geometry that a pfSeg intersects with. The structure definition is included at this time for completeness. A more thorough discussion will be given on the data actually used in Chapter IV.

```
typedef struct {
   long flags;          /* intersection and structure validity */
   long segnum;         /* the number of the segment that has intersected*/
   pfSeg seg;           /* description of segment that has intersected */
   pfVec3[4]  point;    /* point of intersection */
   pfVec3     norm;     /* normal at intersection point */
   long tri;            /* index of triangle in the primitive */
   long prim;           /* index of primitive in the geoset */
   pfGeoSet* gset;      /* pfGeoSet of intersection */
   void* userdata;      /* data for user in callback */
   pfMatrix[5]  xform;  /* transformation to word coordinates */
} pfIsect;
```

7. pfPlane

A pfPlane is a structure which describes a non-graphical, 2-D, infinite plane which is used for intersection testing. A pfPlane is created using a point and a normal:

pfMakeNormPtPlane(pfPlane *dst, pfVec3 norm, pfVec3 pos)

or it is created using three points which will define any plane:

---

4. A pfVec3 is an array of 3 floats
5. A pfMatrix is a 4X4 array of floats.

*pfMakePtsPlane (pfPlane *dst, pfVec3 pt1, pfVec3 pt2, pfVec3 pt3)*

8. pfCylinder

A pfCylinder is a non-graphical cylindrical volume used for intersection testing. It is defined as:

```
typedef struct {
    pfVec3  center;
    float   radius;
    pfVec3  axis;
    float   halfLength;
} pfCylinder;
```

A pfCylinder can be constructed to encompass a group of pfSegs. This allows for a more rapid intersection determination since the pfSegs only intersect the geometry if the pfCylinder does. A single test with the cylinder can be performed prior to testing individual segments. Figure 4 shows a pfCylinder around 4 pfSegs.

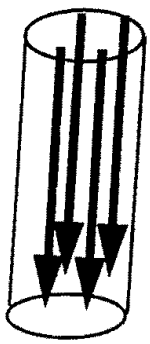


Figure 4: A pfCylinder around pfSegs

The pfCylinder is created using:

*pfCylAroundSegs(pfCylinder *dst, pfSeg **segs, long nseg)*

where nseg is the number of segments to be encompassed by the cylinder. A maximum of 32 pfSegs may be defined.

C. COORDINATE SYSTEM

Performer utilizes an orthogonal coordinate system with the X axis out of the screen, the Y axis to the right and the Z axis up. This differs from the conventional robotics coordinate system where the X axis is North (right), the Y axis is East (out of screen) and the Z axis is down.

In order to ease the incorporation of the motor dynamics simulation [Ref. 10] and the hydrodynamic simulation [Ref. 11] with the graphical simulation, it was necessary to transform the graphics coordinate system into the robotics coordinate system. This can be done in two ways. The first is to manipulate the view volume [Ref. 10] which changes the world coordinate system. Doing this, makes the simulation difficult to combine with other graphics simulations and thus minimizes its portability.

The second method is to isolate the Aquarobot simulation from the graphics environment. This allows the control algorithms to perform joint rotations and center of body translations and rotations using conventional robotics notation while allowing the robot to move in a conventional graphics world.

1. Joint Axis Notation

In robotics, there is also a set notation for the manipulation of articulated links. The X axis is the common normal from the inboard joint axis (closest to center of body) to the outboard joint axis. The Z axis is along the axis of rotation for the inboard joint and the positive direction is chosen to be which ever is beneficial to the implementor. The Y axis is orthogonal to the X and Z axes using the conventional right hand notation (Figure 5).

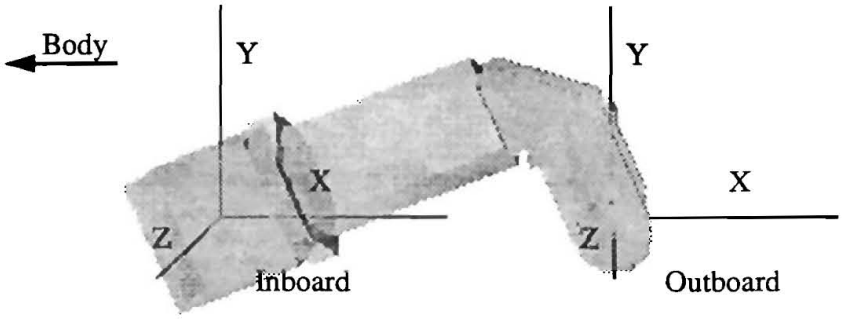


Figure 5: Robotics Joint Axis Notation

D. *AQUAROBOT* CONSTRUCTION

Aquarobot consists of a cylindrical torso, six articulated legs, an articulated camera boom, and a 100 meter tether cable which connects the robot to a support vessel (not shown). Each leg consists of four links, a shoulder, an upper leg, a lower leg, and a foot pad. The camera boom consists two links attached to a rotating base at the top of the torso (Figure 6).

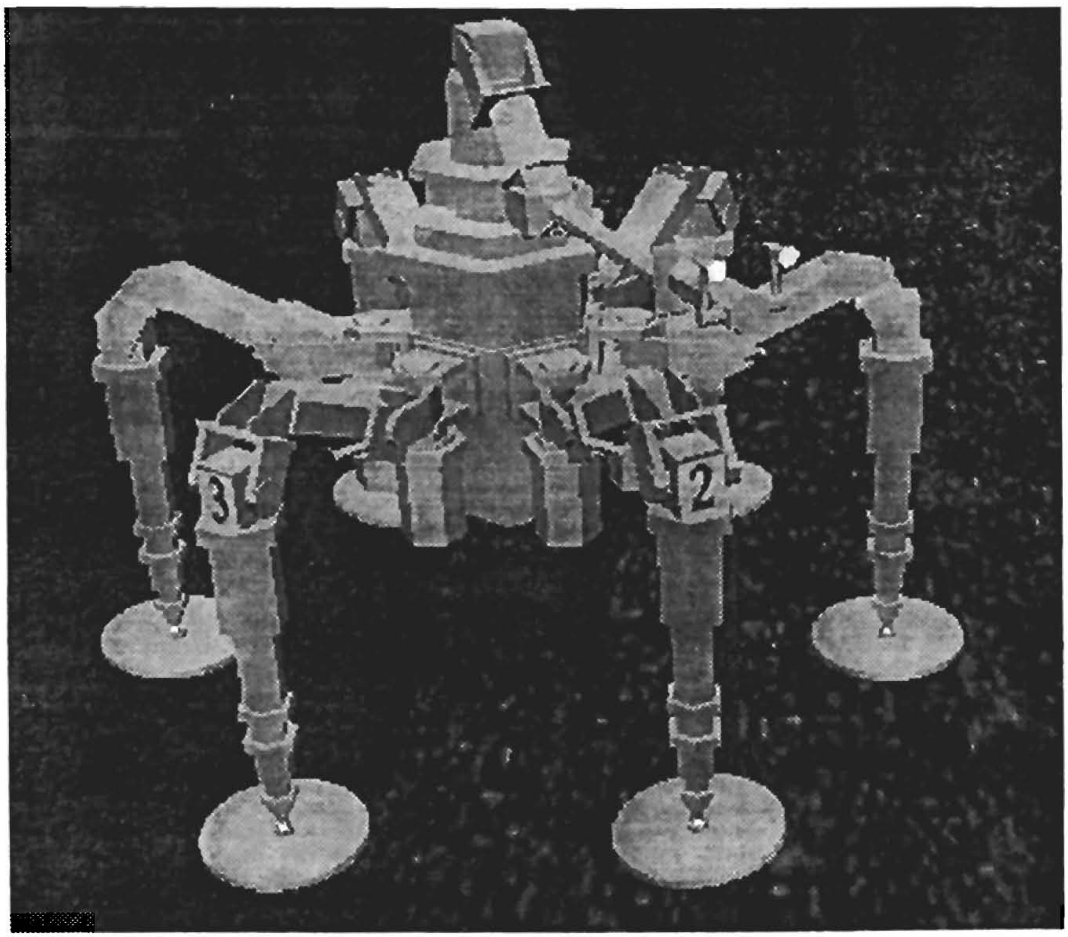


Figure 6: Graphical Representation of *Aquarobot*

The Aquarobot model was constructed by hand using technical drawings and photographs as a guide. Each articulated piece was broken down into a set of polygons and

then stored in a visual database, NPSGDL. Performer was then used to define an articulated system in which to display the models.

1. Performer Node Structure

Depicted in Figure 7, is the tree structure for the torso and one articulated leg of

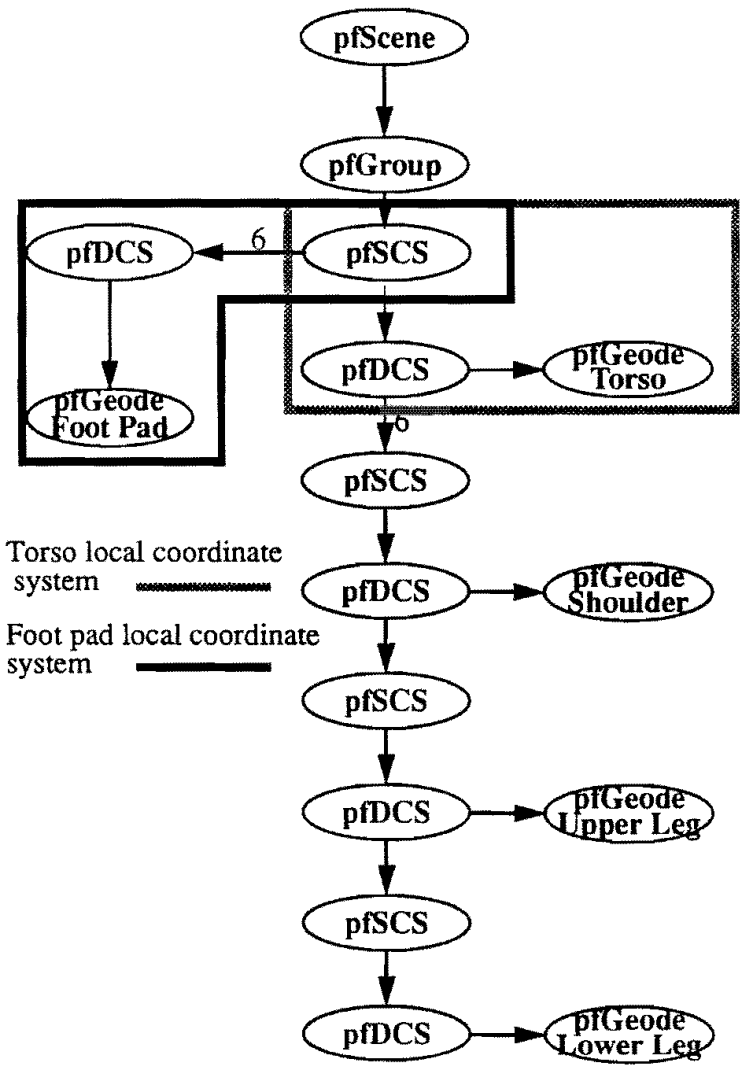


Figure 7: Performer Node Structure for *Aquarobot*

Aquarobot. The tree is comprised of static coordinate systems (pfSCS), dynamic coordinate systems (pfDCS) and pfGeodes, which contain the models for each of the five

different articulated pieces.[6] The numbers adjacent to arcs indicate the number of branches being represented. All nodes on a branch are duplicated the indicated number of times except for the pfGeodes. Models need only be stored once and may be referenced many times.

a. Articulations

When constructing articulated links, the pfSCS is necessary to translate and/or rotate the local coordinate system from joint to joint in order to achieve a coordinate system at the next joint in succession that is oriented with the Z axis coincident with the axis of rotation (Figure 8). The pfDCS is then used to perform a single Z axis rotation which

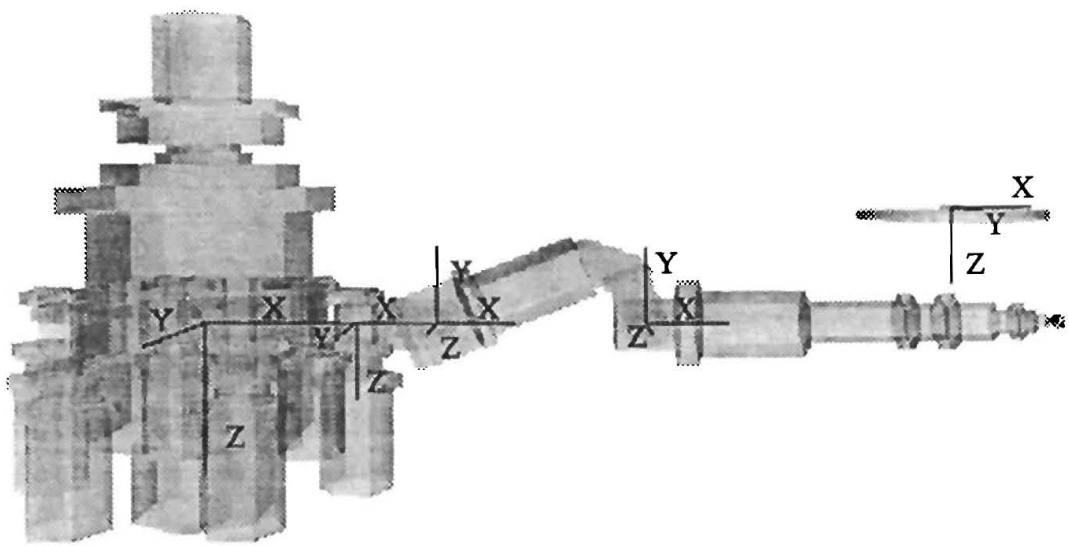


Figure 8: Coordinate Systems for Torso and a Typical Leg With Foot Pad

6. The camera boom was omitted for simplicity

will place the leg in the desired orientation. Figure 9 shows the orientation of the local coordinate systems after being rotated.

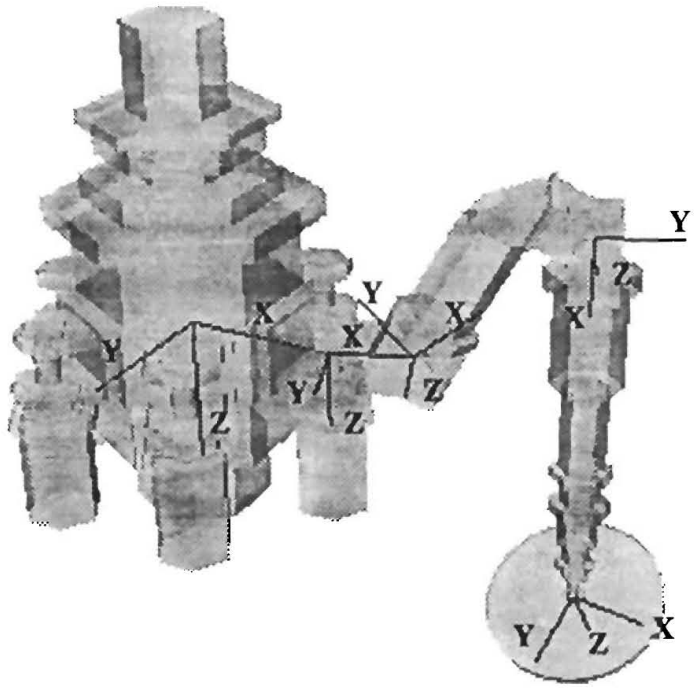


Figure 9: Local Coordinate Systems After Being Rotated

b. Torso Local Coordinate System

The top pfSCS establishes the correct orientation for the torso's local coordinate system. It is achieved by performing two successive rotations; a 90 degree rotation about the Z axis and a 180 degree rotation about the Y axis, and performing a single matrix multiplication. This transforms Performer's coordinate system into a robotics coordinate system. The torso pfDCS is then used to translate and rotate *Aquarobot* throughout the graphics environment.

c. Foot Pad Local Coordinate System

The foot pad coordinate system is unique in that the position is determined by the orientation of the previous links and the orientation is determined by the terrain

characteristics. If the coordinate system was linked to the lower leg, all previous rotations would need to be removed from foot's pfDCS in order for the foot orientation algorithm to place the foot correctly (the algorithm assumes a non rotated coordinate system initially). So in order to maintain an initial non-rotated coordinate system for the feet, it was decided to attach the coordinate system to the world. This does have a potential drawback. Since the foot 's position is based on the world coordinate system, the foot position calculated from the gait algorithm must be transformed to world coordinates. The simplistic gait algorithm used in this research was originally designed in world coordinates, so this transformation was not necessary.

2. Joint Limits

Each limb has a physical joint limit which prevents the limbs from bending too far. The control software has a joint limit also, which is used to try and prevent the use of the physical stops. To ensure the new control software is performing correctly, physical stops in the graphical model were not incorporated.

a. Ball Joints

The ball joint has a physical limit as well. Unlike the other joints, when this limit is reached, the foot pad will begin to tilt in the direction the leg is going (Figure 10).

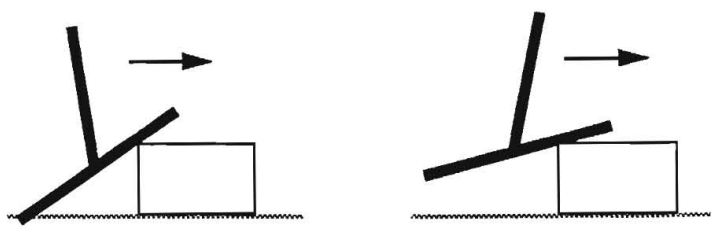


Figure 10: Motion of Foot When Joint Limit is Reached

This limit is not controlled by the gait algorithm since it is determined by both the foot orientation on the ground and the leg position. Reaching this limit causes the foot position to change which then needs to be relayed to the gait algorithm so a new leg position can be calculated. This feedback is beyond the scope of this research and is left for future work.

IV. ENVIRONMENT MODELING FOR FOOT PLACEMENT

The first Aquarobot simulation was limited to traversing on flat terrain. The moving feet continued to move until they reached a known plane, then the next set of feet would be moved. The actual robot has sensors in the ball joint of each foot to signal ground contact. When a foot touches ground, and is exerting enough force to support the robot, the control software knows to stop the vertical motion of that foot and can continue on with the gait pattern. The first step in trying to simulate the functionality of the foot sensors is to establish a terrain on which to walk.

A. TERRAIN MODEL

There are many different types of terrain throughout the world, many of which are only traversable by some sort of legged creature. Since an exhaustive model displaying all the types of terrain is difficult to achieve, not to mention being beyond the scope of this thesis, a simplified terrain representing certain features was designed to test the foot placement algorithm.

1. Characteristics

Figure 11 illustrates the different characteristics portrayed in the terrain model. The foot pad shows the relative size of a foot compared to that of the terrain. Each of these terrain types have unique characteristics which must be overcome by the foot placement

algorithm (Discussed in Chapter V). The rolling hills model is the most simplistic and was used to get the orientation portion of the algorithm correct.

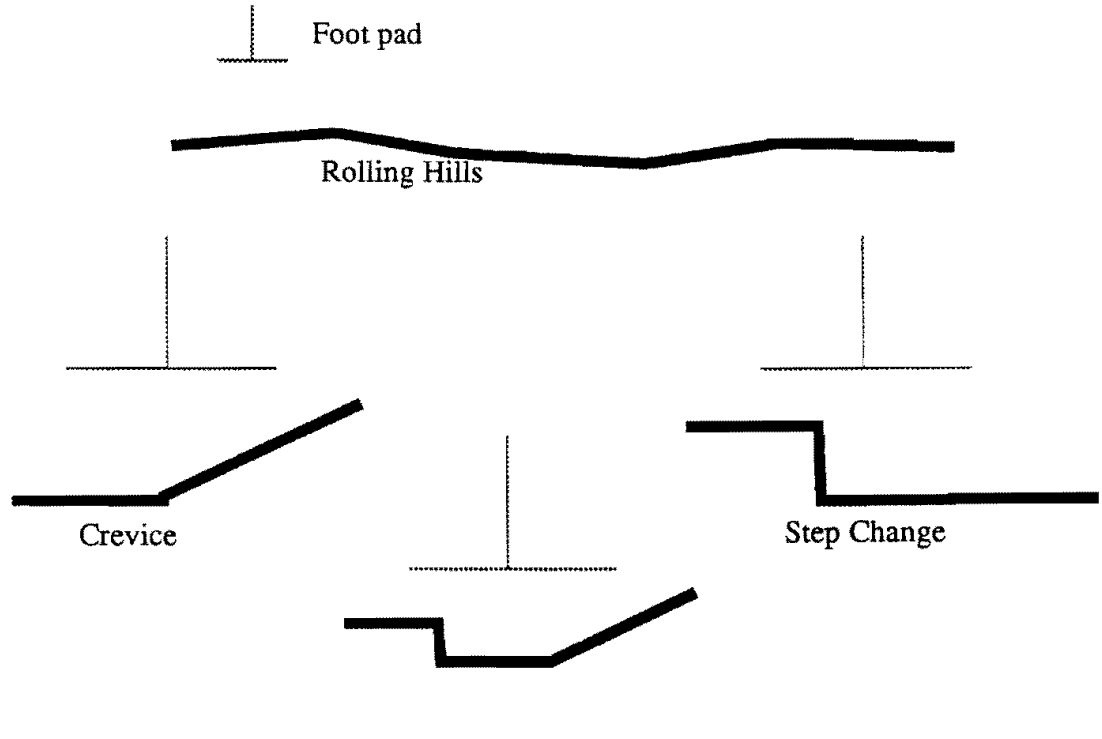


Figure 11: Different Terrain Types Used in Terrain Model

B. LIMITATIONS DUE TO GAIT ALGORITHM

The simplistic gait algorithm used while testing the simulation had some strict limitations imposed upon it since it was originally designed for flat terrain. The vertical range of the foot is 0 to 20 cm, so the terrain must be modeled within this range. The torso's height is fixed. If the terrain rises, the upper leg's joint angle will increase and the torso will be closer to the ground.

C. TERRAIN IDENTIFICATION

A useful feature provided by Performer is the ability to assign a unique 16 bit hexadecimal identification number to each node. This ID, known as a traversal mask, is set by the Performer function:

pfNodeTravMask (pfNode *node, long which, ulong mask, long setMode, long bitOp)

The node specifies which node in the scene to begin with when determining how to assign the traversal mask. The which value identifies the traversal type that the mask is for. The setMode indicates if the mask is to be assigned to the node, the node's descendents or both. The bitOp is used to set the node's mask, or change a pre-existing one.

When a traversal is executed (either draw, cull, or intersect), a bitwise 'AND' is performed between the mask of the traversing function (Chapter V, Section B), and the traversal mask of the terrain. If the result is non-zero, the traversal is continued to the nodes descendents. If the result is zero, the branch is pruned and the traversal continues on to a sibling, if one exists. shows the results of distinguishing between different terrain types.

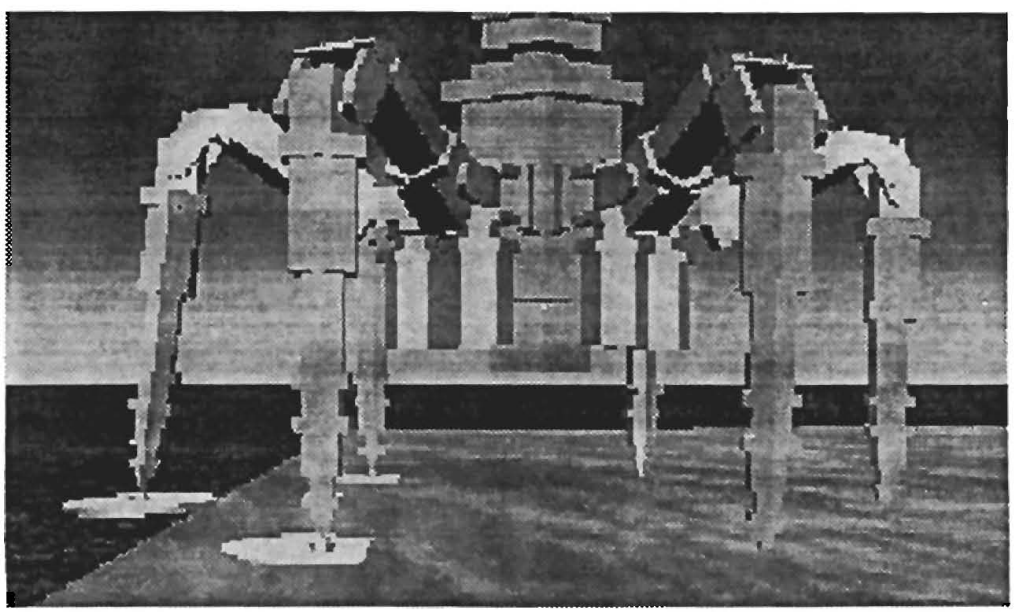


Figure 12: Aquarobot Standing in Water

D. UNDERWATER ENVIRONMENT

The undersea environment in which Aquarobot operates in can be simulated using Performer's environment effects function for fog. Though this gives a fairly accurate impression of being underwater, this effect is only used for demonstration purposes. The fogging effect places a serious drain on the hardware and slows the simulation

E. RESULTS

The limitations on the terrain traversed in the simulation has no bearing on the functionality of the foot placement algorithm to follow. Terrain limitations are a result of limitations imposed on the gait algorithm. The foot placement algorithm is generic in nature to work with any gait pattern.

V. GROUND CONTACT DETERMINATION

Performer has two features which significantly enhance the ability to determine when two graphical objects come in contact with each other. These are bounding volumes and intersecting rays.

A. BOUNDING VOLUMES

A bounding volume is an imaginary space which encompasses an object (Figure 13).

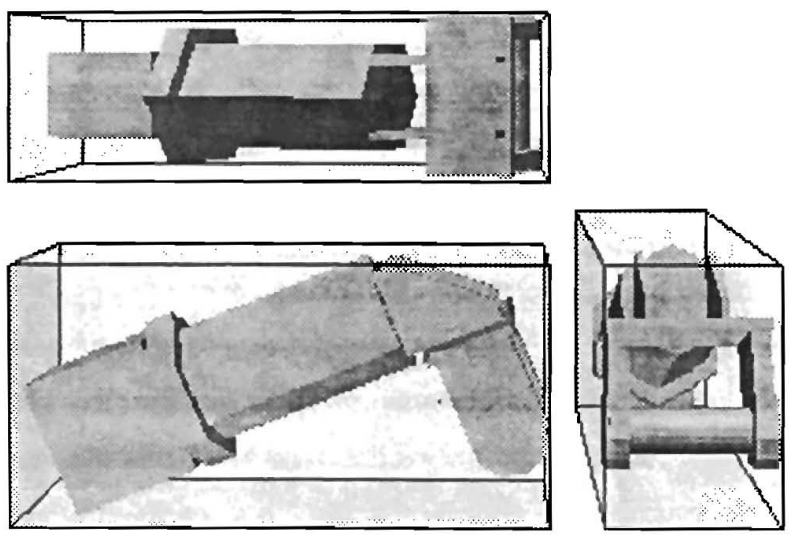


Figure 13: Bounding Box for an Upper Leg

Performer has five different types of bounding volumes, these are boxes, spheres, frustrums, half space, defined by a pfPlane (Chapter III, Section B), and cylinders.

When an intersection test is performed between two bounding volumes, the results are:

- No intersection
- Partial intersection
- Complete intersection: One volume completely inside the other

These limited responses do not allow for the positioning of one volume against another, so volumes are inadequate for the orientation of the foot pads on uneven terrain.

Another drawback to using bounding volumes is that they must be axially aligned. If the object's coordinate system is not coincident with the world coordinate system, the bounding volume does not give a true representation of the bounded object.

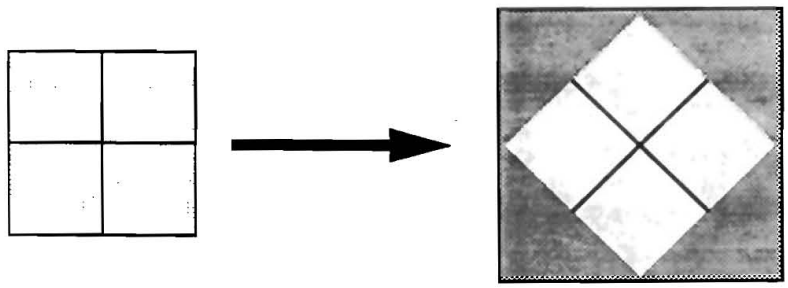


Figure 14: Growth of a Bounding Volume

Figure 14 illustrates the expansion of a bounding volume when the object it encompasses is rotated 45 degrees. Since the bounding volume grew from one square inch to two square inches, half of the intersections with this object will be false (solid grey area represent false collisions). These are unacceptable results in a visual simulation.

B. INTERSECTING SEGMENTS

As previously mentioned, Performer utilizes pfSegs, non-graphical line segments, to test the scene geometry during the intersection traversal of the node structure. These segments are projected from a desired position for a given direction and distance. Intersection testing is accomplished using the Performer command:

pfSegsIsectNode(pfNode *node, pfSeg **segs, long nseg, long mode,
ulong mask, pfCylinder *bcyl, pfIsect *isects,
long (*discFunc)(pfIsect *isect))

The traversal begins at the specified pfNode, and all descendents are checked for intersection with the nseg number of pfSegs. Performer allows up to 32 pfSegs to be specified for any intersection routine. To reduce traversal times, a pfCylinder is specified which encompasses the pfSegs (see Chapter III, Section B).

For each pfSeg which successfully intersects the geometry, the intersection traversal returns information which is stored in an array of nseg pfIsect structures (see Chapter III, Section B). The only information currently being used is the terrain height (Z position in local coordinates) and the transformation matrix. This matrix is necessary to convert the height information to world coordinates so it may be used by the foot orientation algorithm.

The mask is used to identify which type of geometry the intersection test is for. This allows *Aquarobot* to distinguish between stepping on hard ground and water, and perform accordingly.

The ***discFunc*** is a discriminator callback function which is invoked upon all successful intersections. This is a user defined function which provides a more powerful means to control intersections than with the mask test alone. Currently, this option is not being used.

Performer has provided the desired tools to extract the necessary information from any graphics scene. We must now determine how best to utilize these tools to obtain a realistic simulation that runs in real time.

1. A Single Center Segment

The first attempt at getting the foot pads to contour to uneven terrain was to project a single segment from the center of the foot in the -Z direction (that's +Z in the conventional robotics coordinate system) and determine the XYZ coordinate and the normal **P** at the point of intersection. This data was then used to set the height and orientation of the foot pad (Figure 15).

a. Results

This method was very rapid and worked well when the terrain was designed with large sloping primitives (polygons) and the normals between two adjacent primitives did not differ very much. When this method was used with step changes in height, the visual results were displeasing (Figure 16).

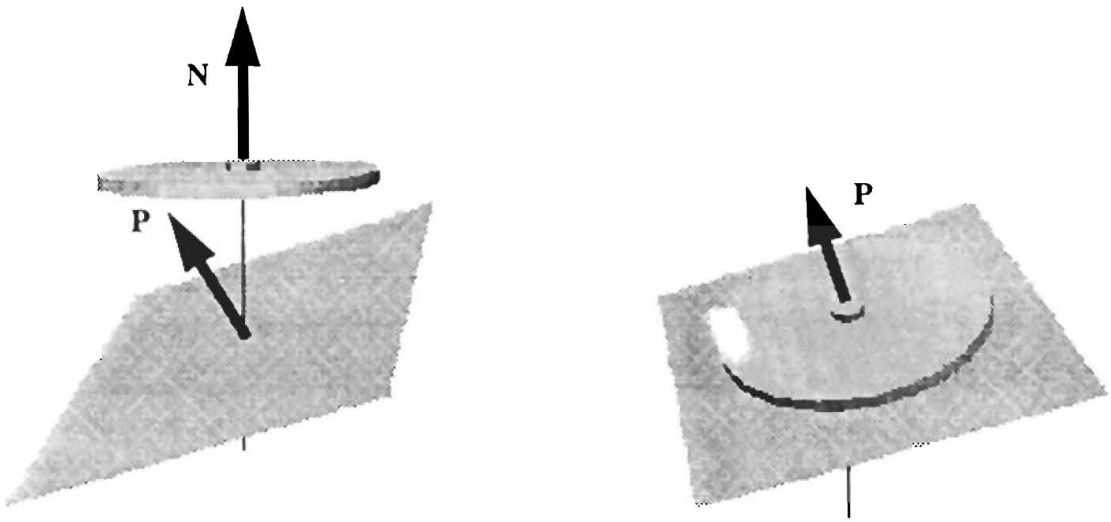


Figure 15: Foot Placement With a Single Segment at the Center of the Foot

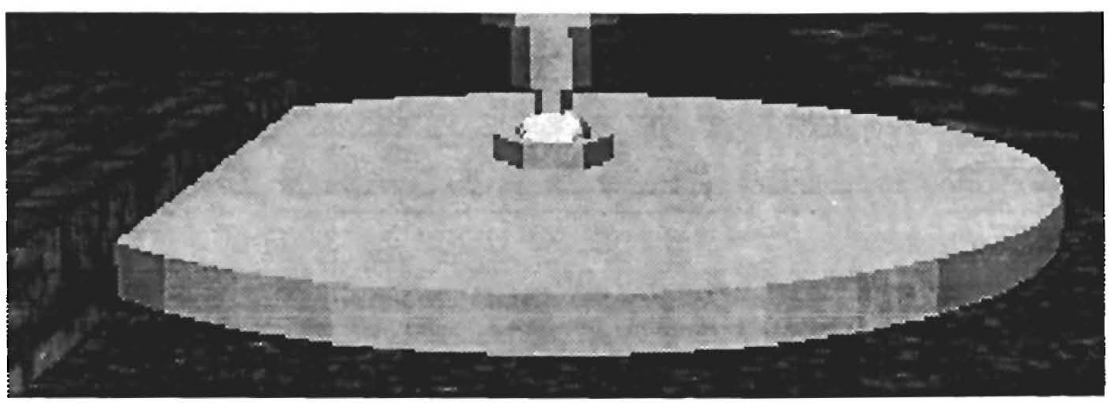


Figure 16: Result of Using a Single Segment From the Center of the Foot

2. Multiple Segments

The next approach in determining foot placement evolved from the first. Multiple intersecting segments were organized in a manner to extract more information from the terrain while taking maximum advantage in the use of Performers pfPlane (Chapter III, Section B).

a. Foot Pad Divisions

The intersection segments (pfSegs) are arranged in three groups which divide the foot pad into equal sectors of 120 degrees. Figure 17 shows the general pattern used for the placement of the intersecting segments.

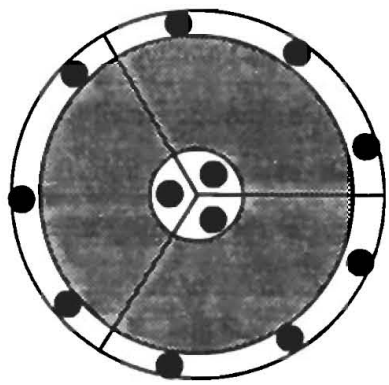


Figure 17: Placement of Segments

This pattern allows for the 100% detection of objects which are greater than 20 cm in diameter. Objects less than 20 cm may go undetected by the simulation if stepped on in just the right manner. The shaded area of Figure 17 depicts the "dead zone" of the foot pad where objects will be undetected.

b. Creating an Artificial Plane

Dividing the foot pad into 3 sectors was chosen on the concept that any 3 non-linear points define a unique plane. If the highest identified point from each sector is used, a plane can be defined on which to place the foot. Thus the orientation problem is

reduced down to the single segment solution. This approach eliminates the problem shown in Figure 16, but is still not adequate enough for an accurate simulation (Figure 18).

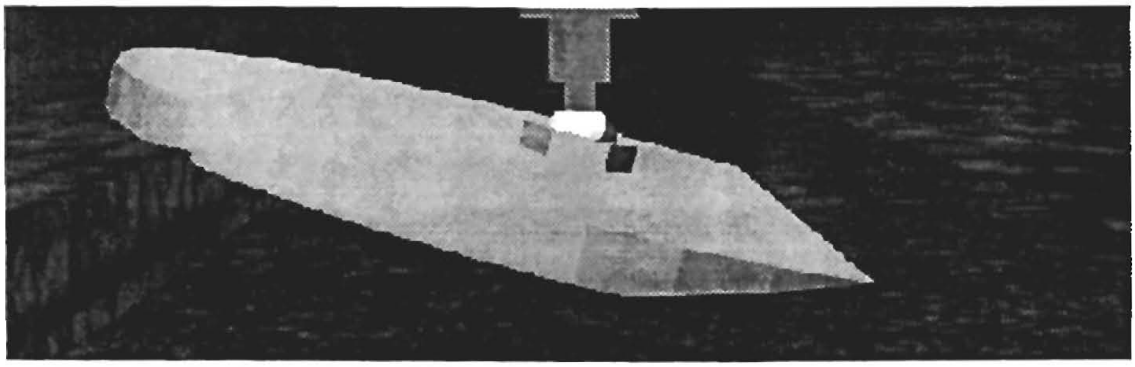


Figure 18: Foot Orientation Using an Artificial Plane

c. Correcting the Artificial Plane

Once an artificial plane has been established, it is necessary to examine the foot pad oriented to this plane and check for collisions with the surrounding geometry. If such collisions exist, then it is necessary to recompute the artificial plane. Here is a description of the algorithm used for foot placement on uneven terrain:

- Determine the highest point in each sector of a foot pad.
- Create a pfPlane (Chapter III, Section B) using the three highest points.
- Determine the height of the pfPlane at the center of the foot pad.
- Set the foot pad to conform to the pfPlane height and orientation.
- Calculate the height at each of the locations used to project the pfSegs based on the new orientation.
- Compare the height at each location with the actual ground height returned from the intersection test.
- For each sector, if the ground height is greater than the foot pad height, find the point with the greatest difference and substitute this point into the pfPlane equation to determine a new plane.
- Set the foot pad to the corrected pfPlane orientation.

d. Results

After a single iteration of adjusting the ground plane, the simulation was able to overcome the previous two anomalies and produce a fairly accurate simulation

(Figure 19). But this method too has its drawbacks. When traversing terrain that contains

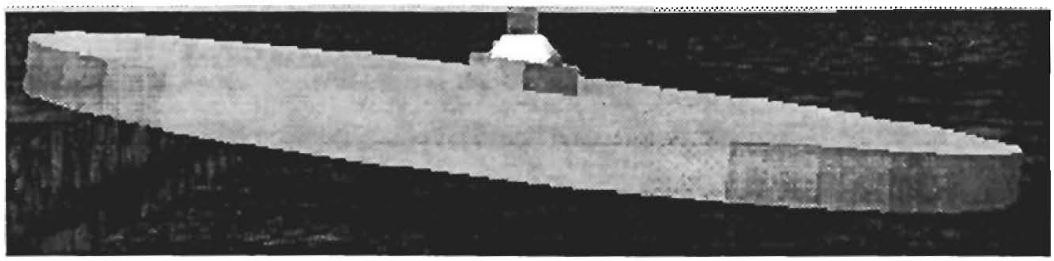


Figure 19: Final Foot Placement

raised obstacles that lie perpendicular to one of the lines that divide a foot pad, a unique situation arises if the three contact points for determining the artificial plane happen to fall on one half of the foot pad (Figure 20).

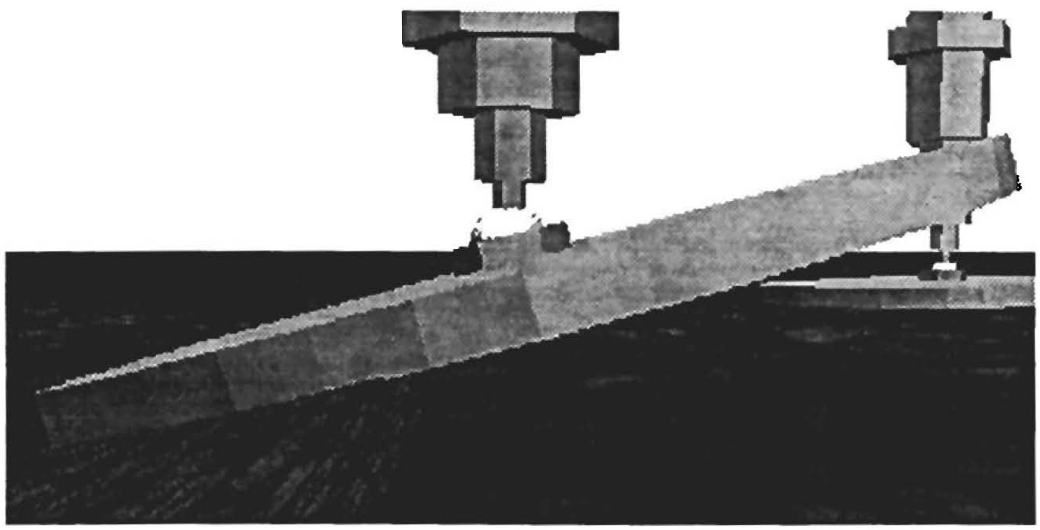


Figure 20: Result of Having all Three Ground Contact Points on One Side of the Foot

Though, this feature is not visually or physically correct, it does not invalidate the simulation as a tool for testing the control software. The visual simulation still sets the foot's ground contact flag to inform the gait algorithm to stop moving the foot independent of the foot's orientation.

3. Speeding Up the Process

Other than defining a pfCylinder to speed up the intersection traversal of the terrain node, two flags were added which reduce the number of times the intersection traversal is performed. The first is a ground contact flag. If the foot is on the ground, then the routine for foot placement is not necessary. The second flag defines foot vertical motion. Checking to see if the foot is touching the ground is only necessary if the foot is being lowered. These two flags caused the simulations execution time to increase from 15 frames per second (FPS), to 30 FPS.

VI. RESULTS

A. SIMULATION DESCRIPTION

The Aquarobot simulation is a real-time, 3-D computer simulation running on a Silicon Graphics Workstation. The Simulation is currently running at approximately 30 frames per second. Additional hardware utilized is a Spaceball[1] for the position of the viewing volume, SGI's Dial and Button indicators for the control of the camera boom, and HOTAS flight control devices for *Aquarobot's* center of body commanded velocity (velocity may also be controlled using the keyboard, See Below). Software needed for the simulation is SGI's Performer, for its rapid drawing and intersecting routines and the NPSGDL loader which enables Performer to display the models.

1. Keyboard Responses

The following describes the keyboard responses recognized by the simulation:

- ESC key - exits from the simulation
- F1 key - Displays system characteristics
- F2 key - Enables or disables texturing
- F3 key - Toggles fog on/off (underwater effect)

The arrow keys control the velocity of the torso. The responses are 0% to 100% of the maximum simulation speed in increments of 20%.

1. A spaceball is a 6 degree torque sensing device

VII. CONCLUSIONS AND RECOMMENDATIONS

A. RECOMMENDATIONS

Work is being done to create a virtual environment for AUV's [Ref. 14] to test hardware and control software performance in a controlled, yet realistic manner. This same theory may be applied to walking machines. This thesis demonstrated the ability to distinguish between solid ground and water, the two extremes. It is also possible to model the terrain to exhibit certain characteristics in between and also to model foot sinkage and slippage. I believe that the development of a virtual environment that has the capability of providing foot sinkage and foot slippage information in the form of reaction forces to the placement of the foot will enhance the development of fully autonomous walking robots.

B. FUTURE WORK

This thesis is just the beginning of creating a complete simulation for the testing of control software. Other areas that need research are:

- Detecting collisions between limbs
- Static forces acting on Aquarobot from the tether cable
- Modeling the ball joint limit and the feedback to the gait algorithm

Another area of research is the integration of walking robots into defense simulations to investigate their use in mine hunting operations.

APPENDIX A: SIMULATION SOURCE CODE

```
// ***************************************************************************
// FILENAME:    pfaqua.C
// PURPOSE:     This file contains the functions to set up the geometry for
//              the Aquarobot model, determining ground contact and determining
//              foot pad orientation
//
// NOTE:        This program is written in C++ utilizing Performer 1.0
// AUTHOR:      John Goetz
// DATE:        20 February 1994
// COMMENT:
// UPDATE:
// ***************************************************************************

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <gl/device.h>

#include "SGIwindowcam.H"
#include "robot_globals.h"
#include "loadGDL.h"
#include "pf.h"

#define TERRAIN_MASK  0x0008
#define WATER_MASK    0x0004
#define RAYS_PER_SECTOR 10
#define FOOTPAD_RADIUS  22.5f

static void OpenPipeline (pfPipe *p);
static void DrawChannel (pfChannel *channel, void *data);
static void CullChannel(pfChannel *chan, void *data);

void ground_contact(pfDCS *DCSlink[6][4]);
void set_initial_foot_position(pfDCS *DCSlink[6][4]);
void set_foot_intersection_segments(pfSeg **segment, int leg);
void orient_foot(pfIsect data[3*RAYS_PER_SECTOR+1], int leg);
void move_camera_boom(pfDCS *DCSboom[3]);
void read_materials();
pfGroup* read_terrain();

pfGroup* build_aquarobot(pfDCS *base, pfDCS *DCSlink[6][4], pfDCS *DCSboom[3]);


static pfScene *scene;
static pfLight *Sun, *Sun2;
static pfFog   *water;

pfSeg   **segment = (pfSeg**)pfMalloc(sizeof(pfSeg*) * (3 * RAYS_PER_SECTOR),NULL);

static SharedData *Shared;
```

```
void main() {

   pfPipe     *p;
   pfChannel  *chan;
   pfGroup    *aquarobot1;
   pfGroup    *terrain;
   pfDCS      *aquabase;
   pfDCS      *DCSlink[6][4];
   pfDCS      *DCScamboom[3];
   pfEarthSky *eSky;


// initialize Performer
   pfInit();

// Allows multiple threads to be forked. One for the application, one for
// the Draw process and one for the Cull process
   pfMultiprocess(PFMP_DEFAULT);

// This prevents multiple threads from being forked.
// pfMultiprocess(PFMP_APPCULLDRAW);

// Establish shared memmory before calling pfConfig() so that all forked
// threads will know where it is.

   Shared = (SharedData*)pfMalloc(sizeof(SharedData), pfGetSharedArena());
   Shared->exitFlag  = 0;
   Shared->resetFlag = 0;
   Shared->pfStats   = 0;
   Shared->pfFog     = 0;

   pfConfig();

   scene = pfNewScene();

   read_materials();

// Load in the terrain model for robot to walk on and add it to the scene as
// the first child (index 0). This enables the terrain to be accessed redily
// by the ground contact algorithms.

   terrain = read_terrain();
   pfAddChild (scene, terrain);

// Build the Aquarobot model
   aquabase   = pfNewDCS();
   aquarobot1 = build_aquarobot(aquabase, DCSlink, DCScamboom);

// Add the robot to the scene
   pfAddChild (scene,aquarobot1);


// Configure and open a graphics pipeline
   p = pfGetPipe(0);
   pfInitPipe(p, OpenPipeline);

// Set desired frames per second
   pfFrameRate(30.0f);
```

```
// Get and configure a viewing channel
   chan = pfNewChan(p);

// Set the channels Draw and Cull functions
   pfChanDrawFunc(chan, DrawChannel);
   pfChanCullFunc(chan, CullChannel);

// Establish dimentions for the viewing frustrum
   pfChanScene(chan, scene);
   pfChanNearFar(chan, 0.1f, 6000.0f);
   pfChanFOV(chan, 45.0f, -1.0f);

// Establish orientation for viewing frustrum
   pfSetVec3(Shared->view.hpr, 90.0f, 0.0f, 0.0f);
   pfSetVec3(Shared->view.xyz, 650.0f, 0.0f, 20.0f);

// Create an earth/sky model that draws sky and horizon
   eSky = pfNewESky();
   pfESkyMode(eSky, PFES_BUFFER_CLEAR, PFES_SKY_CLEAR);
   pfChanESky(chan, eSky);

//   pfChanTravMode(chan, PFCULL_VIEW|PFCULL_GSET, NULL);

// This pfSync and pfFrame are necessary to establish the graphics scene so
// that the pfSegsIsectNode routine can be used in set_initial_foot_position.

   pfSync();
   pfFrame();

// This restart gets the X and Y positions of the footpads
   restart_robot(aquabase, DCSlink);

// The Z position of the footpads is set based on terrain information
   set_initial_foot_position(DCSlink);

// Move robot is called so that the joint angles may be calculated based
// on the correct foot height
   move_robot(aquabase, DCSlink);

// Start the continuous loop
   while (!Shared->exitFlag) {


     /* Go to sleep till next frame time */
     pfSync();

     /* Set view parameters. */
     pfChanView(chan, Shared->view.xyz, Shared->view.hpr);

     /* initiate cull/draw for this frame */
     pfFrame();

     /* Update robot position */
     ground_contact(DCSlink);
     move_robot(aquabase, DCSlink);
     move_camera_boom(DCScamboom);


     if (Shared->resetFlag) {
       pfSetVec3(Shared->view.hpr, 90.0f, 0.0f, 0.0f);
       pfSetVec3(Shared->view.xyz, 650.0f, 0.0f, 20.0f);
```

```
        restart_robot(aquabase, DCSlink);
        set_initial_foot_position(DCSlink);
        Shared->resetFlag = 0;
      }

  }

  /* terminate cull and draw processes (if they exist) */
  pfExit();

  /* exit to operating system */
  exit(0);

}


//   OpenPipeline() -- create a pipeline: setup the window system,
//   the IRIS GL, and IRIS Performer. This procedure is executed in
//   the draw process (when there is a separate draw process).


static void
OpenPipeline (pfPipe *p)
{

  /* negotiate with window-manager */
  scrnselect(0);
  foreground();

  /* set up window */
  prefposition(178, 779, 491, 952);
  winopen("Aquarobot");

  /* negotiate with GL */
  pfInitGfx(p);

  /* Set Z-buffer depth based on hardware */
  if ( int(getgconfig(GC_MS_SAMPLES)) > 0 )
    lsetdepth ( getgconfig(GC_MS_ZMIN), getgconfig(GC_MS_ZMAX) );
  else
    lsetdepth ( getgconfig(GC_ZMIN), getgconfig(GC_ZMAX) );

  /* initialize events to be placed on the que */
  initialize();

  /* create two light sources */
  Sun  = pfNewLight(pfGetSharedArena());
  Sun2 = pfNewLight(pfGetSharedArena());
  pfLightPos(Sun, 0.5f, 0.5f, 1.0f, 0.0f);
  pfLightPos(Sun2, 0.5f, -0.5f, 1.0f, 0.0f);

  /* create a default lighting model */
  pfApplyLModel(pfNewLModel(pfGetSharedArena()));

  pfLightOn(Sun);
  pfLightOn(Sun2);

  /* enable culling of back-facing polygons */
  pfCullFace(PFCF_BACK);
```

```
    /* Set up fog parameters to simulate an underwater environment */
    pfEnable(PFEN_FOG);
    water = pfNewFog(pfGetSharedArena());
    pfFogType(water, PFFOG_PIX_EXP);
    pfFogColor(water, 0.0f, 0.385f, 0.464f);
    pfFogRange(water, 0.0f, 50000.0f);

}

/*
 *    CullChannel() -- traverse the scene graph and generate a
 *    display list for the draw process.  This procedure is
 *    executed in the cull process.
 */

static void
CullChannel(pfChannel*, void*) {

    /*
     * pfDrawGeoSet or other display listable Performer routines
     * could be invoked before or after pfCull()
     */

    pfCull();
}


static void DrawChannel (pfChannel *chan, void*) {

//    pfCoord *temp = (pfCoord*)data;
      static pfVec4 backdrop;
//    pfSetVec4(backdrop, 0.0f, 0.385f, 0.464f, 1.0f);
    pfSetVec4(backdrop, 1.0f, 1.0f, 1.0f, 1.0f);
    pfClear(backdrop, PFCL_CLEARCZ);
    /* rebind light so it stays fixed in position */
    pfLightOn(Sun);
    pfLightOn(Sun2);


    /* draw Performer throughput statistics */
    if (Shared->pfStats)
      pfDrawChanStats(chan);

//    if (Shared->pfFog)
      pfApplyFog(water);
//    else
//      pfClearChan(chan);

// Invoke Performer draw-processing for this frame
    pfDraw();

    Check_Que(Shared);

} // end DrawChannel
```

```
// Build_aquarobot sets the DCS's so that all the legs are connected.
// This does not set the initial posture

pfGroup* build_aquarobot(pfDCS *base, pfDCS *DCSlink[6][4], pfDCS *DCSboom[3]) {

  register int i;
  pfGroup *dummy;
  pfGroup *shoulderH, *shoulderM;
  pfGroup *upper_legH, *upper_legM;
  pfGroup *lower_legH, *lower_legM;
  pfGroup *foot_pad;
  pfGroup *legnumber[6];

  pfSCS  *shoulderSCS[6],
       *upper_legSCS[6], *lower_legSCS[6],
       *camera_boomSCS[3],
       *scs1;

  pfLOD     *torsoLOD, *shoulderLOD,
          *upper_legLOD, *lower_legLOD;

  pfMatrix   rot_mat, rot_mat2, trans_mat;

  dummy = pfNewGroup();

  /* The torso image is defind with the X axis comming out of the bottom of the
    robot (this is the gl Z axis) so it must be rotated 90 deg about the Y axis
    for it to be oriented properly. */

  pfMakeRotMat(rot_mat, 90.0f, 0.0f, 0.0f, 1.0f);
  pfMakeRotMat(rot_mat2, 180.0f, 0.0f, 1.0f, 0.0f);
  pfPreMultMat(rot_mat2, rot_mat);
  scs1 = pfNewSCS(rot_mat2);
  pfAddChild(dummy, scs1);


  torsoLOD = pfNewLOD();
  pfLODRange(torsoLOD, 0, 1.0f);
  pfLODRange(torsoLOD, 1, 100.0f);
  pfLODRange(torsoLOD, 2, 10000.0f);
  pfAddChild(dummy, base);
  pfAddChild(base, torsoLOD);
  pfAddChild(torsoLOD, LoadGDL2("models/Robotics/pftorsoLODH.gdl"));
  pfAddChild(torsoLOD, LoadGDL2("models/Robotics/pftorsoLODM.gdl"));


  shoulderH = LoadGDL2("models/Robotics/pfshoulder_linkLODH.gdl");
  shoulderM = LoadGDL2("models/Robotics/pfshoulder_linkLODM.gdl");
  upper_legH = LoadGDL2("models/Robotics/pfupper_legLODH.gdl");
  upper_legM = LoadGDL2("models/Robotics/pfupper_legLODM.gdl");
  lower_legH = LoadGDL2("models/Robotics/pflower_legLODH.gdl");
  lower_legM = LoadGDL2("models/Robotics/pflower_legLODM.gdl");
  foot_pad   = LoadGDL2("models/Robotics/pffoot_pad.gdl");

  legnumber[0] = LoadGDL2("models/Robotics/numplate1.gdl");
  legnumber[1] = LoadGDL2("models/Robotics/numplate2.gdl");
  legnumber[2] = LoadGDL2("models/Robotics/numplate3.gdl");
  legnumber[3] = LoadGDL2("models/Robotics/numplate4.gdl");
  legnumber[4] = LoadGDL2("models/Robotics/numplate5.gdl");
  legnumber[5] = LoadGDL2("models/Robotics/numplate6.gdl");
```

```
for(i = 0; i<6; i++) {

 /* Rotate and translate shoulder so it will be rendered correctly */
 pfMakeRotMat(rot_mat, (float)i*60.0f, 0.0f, 0.0f, 1.0f);
 pfMakeTransMat(trans_mat, LINK0LENGTH, 0.0f, 0.0f);
 pfPreMultMat(rot_mat, trans_mat);
 shoulderSCS[i] = pfNewSCS(rot_mat);

 /* Add DCS to the SCS so the shoulder can move */
 DCSlink[i][0] = pfNewDCS();
 shoulderLOD = pfNewLOD();
 pfLODRange(shoulderLOD, 0, 1.0f);
 pfLODRange(shoulderLOD, 1, 100.0f);
 pfLODRange(shoulderLOD, 2, 10000.0f);
 pfAddChild(base, shoulderSCS[i]);
 pfAddChild(shoulderSCS[i], DCSlink[i][0]);
 pfAddChild(DCSlink[i][0], shoulderLOD);
 pfAddChild(shoulderLOD, shoulderH);
 pfAddChild(shoulderLOD, shoulderM);

 /* Translate out from shoulder DCS to the next link */
 pfMakeTransMat(trans_mat, LINK1LENGTH, 0.0f, 0.0f);
 pfMakeRotMat(rot_mat, -90.0f, 1.0f, 0.0f, 0.0f);
 pfPreMultMat(trans_mat, rot_mat);
 upper_legSCS[i] = pfNewSCS(trans_mat);
 DCSlink[i][1] = pfNewDCS();
 upper_legLOD = pfNewLOD();
 pfLODRange(upper_legLOD, 0, 1.0f);
 pfLODRange(upper_legLOD, 1, 500.0f);
 pfLODRange(upper_legLOD, 2, 10000.0f);
 pfAddChild(DCSlink[i][0], upper_legSCS[i]);
 pfAddChild(upper_legSCS[i], DCSlink[i][1]); // DCS to move upper leg
 pfAddChild(DCSlink[i][1], upper_legLOD);
 pfAddChild(upper_legLOD, upper_legH);
 pfAddChild(upper_legLOD, upper_legM);

 /* Translate out from upper_leg DCS to the next link */
 pfMakeTransMat(trans_mat, LINK2LENGTH, 0.0f, 0.0f);
 lower_legSCS[i] = pfNewSCS(trans_mat);
 DCSlink[i][2] = pfNewDCS();
 lower_legLOD = pfNewLOD();
 pfLODRange(lower_legLOD, 0, 1.0f);
 pfLODRange(lower_legLOD, 1, 500.0f);
 pfLODRange(lower_legLOD, 2, 10000.0f);
 pfAddChild(DCSlink[i][1], lower_legSCS[i]);
 pfAddChild(lower_legSCS[i], DCSlink[i][2]);// DCS to move lower leg
 pfAddChild(DCSlink[i][2], lower_legLOD);
 pfAddChild(DCSlink[i][2], legnumber[i]);
 pfAddChild(lower_legLOD, lower_legH);
 pfAddChild(lower_legLOD, lower_legM);

 /* Attach foot pad DCS's to the world not Aquarobot*/
 DCSlink[i][3] = pfNewDCS();
 pfAddChild(scs1, DCSlink[i][3]);
 pfAddChild(DCSlink[i][3], foot_pad);

 pfFlatten(dummy);

} // end for statement */

 pfMakeTransMat(trans_mat, 0.0f, 0.0f, -87.0f);
```

```
  camera_boomSCS[0] = pfNewSCS(trans_mat);
  DCSboom[0] = pfNewDCS();
  pfAddChild(base, camera_boomSCS[0]);
  pfAddChild(camera_boomSCS[0], DCSboom[0]);
  pfAddChild(DCSboom[0], LoadGDL2("models/Robotics/pfcamera_base.gdl"));

  pfMakeTransMat(trans_mat, 5.0f, 0.0f, 0.0f);
  pfMakeRotMat(rot_mat, -90.0f, 1.0f, 0.0f, 0.0f);
  pfPreMultMat(trans_mat, rot_mat);
  camera_boomSCS[1] = pfNewSCS(trans_mat);
  DCSboom[1] = pfNewDCS();
  pfAddChild(DCSboom[0], camera_boomSCS[1]);
  pfAddChild(camera_boomSCS[1], DCSboom[1]);
  pfAddChild(DCSboom[1], LoadGDL2("models/Robotics/pfcamera_link1.gdl"));

  pfMakeTransMat(trans_mat, 45.0f, 0.0f, 0.0f);
  camera_boomSCS[2] = pfNewSCS(trans_mat);
  DCSboom[2] = pfNewDCS();
  pfAddChild(DCSboom[1], camera_boomSCS[2]);
  pfAddChild(camera_boomSCS[2], DCSboom[2]);
  pfAddChild(DCSboom[2], LoadGDL2("models/Robotics/pfcamera_link2.gdl"));

 return dummy;
} // end build_aquarobot
```

```
void ground_contact(pfDCS *DCSlink[6][4]) {

 static long       nsegs = 3 * RAYS_PER_SECTOR;
 long   isect;
 static pfIsect    result[3 * RAYS_PER_SECTOR];
 static pfCylinder  *cyl = (pfCylinder*)pfMalloc(sizeof(pfCylinder),NULL);
 pfVec3 head, head90;
 float dotp;
 pfMatrix pitch, roll;

 for (int leg = 0; leg < 6; leg++) {

  if ((!foot_data[leg].ground_contact)  &&
    (foot_data[leg].direction)){

    set_foot_intersection_segments(segment, leg);

    pfCylAroundSegs(cyl, segment, nsegs);
    pfSetVec3(cyl->center, foot_data[leg].foot_xyz[2], foot_data[leg].foot_xyz[0],
              foot_data[leg].foot_xyz[1]);
    cyl->halfLength = LINK4LENGTH;
    cyl->radius = FOOTPAD_RADIUS;
    /* find intersection with terrain */
    isect =
     pfSegsIsectNode(pfGetChild(scene, 0), segment, nsegs, PFTRAV_IS_PRIM|
                PFTRAV_IS_CULL_BACK|PFTRAV_IS_NORM,
                TERRAIN_MASK,
                cyl, result, NULL);

    if ((isect) &&
      (foot_data[leg].foot_xyz[1] - result[0].point[2] < 9.0f)) {
      orient_foot(result, leg);

       pfMakeTransMat(foot_data[leg].foot_mat, foot_data[leg].foot_xyz[0],
           foot_data[leg].foot_xyz[2], -foot_data[leg].foot_xyz[1]);

//      Set heading of foot pad. Always 000
      head[0] = 0.0f;
      head[1] = 1.0f;
      head[2] = 0.0f;
//    Determine foot roll
      dotp = PFDOT_VEC3(head, foot_data[leg].normal);
      pfMakeRotMat(roll, (pfArcCos(dotp) - 90.0f), 0.0f, 1.0f, 0.0f);

//    Set heading of foot pad + 90 degrees
      head90[0] = 1.0f;
      head90[1] = 0.0f;
      head90[2] = 0.0f;
//    Determine foot pitch
      dotp = PFDOT_VEC3(head90, foot_data[leg].normal);
      pfMakeRotMat(pitch, (90.0f - pfArcCos(dotp)), 1.0f, 0.0f, 0.0f);
      pfPostMultMat(pitch, roll);
      pfPreMultMat(foot_data[leg].foot_mat, pitch);

//    Set the foot pad DCS to the calculated orientation
      pfDCSMatrix(DCSlink[leg][3], foot_data[leg].foot_mat);

    } // end if statement (isect)
  } // end if statement (not foot contact)
 } // end for statement
} // end ground_contact
```

```
void set_initial_foot_position(pfDCS *DCSlink[6][4]) {

 long    nsegs = 3 * RAYS_PER_SECTOR;
 long    isect;
 pfIsect       result[3 * RAYS_PER_SECTOR];
 pfVec3 head, head90;
 float dotp;
 pfMatrix pitch, roll;

 for (int leg = 0; leg < 6; leg++) {

  set_foot_intersection_segments(segment, leg);

  /* find intersection with terrain */
  isect =
    pfSegsIsectNode(pfGetChild(scene, 0), segment, nsegs, PFTRAV_IS_PRIM|
                PFTRAV_IS_CULL_BACK|PFTRAV_IS_NORM,
                TERRAIN_MASK,
                NULL, result, NULL);

  if (isect)
   orient_foot(result, leg);

  else {
   foot_data[leg].foot_xyz[1] = LINK4LENGTH;
   PFSET_VEC3(foot_data[leg].normal, 0.0f, 0.0f, 1.0f);
  }

  pfMakeTransMat(foot_data[leg].foot_mat, foot_data[leg].foot_xyz[0],
          foot_data[leg].foot_xyz[2], -foot_data[leg].foot_xyz[1]);


// Set heading of foot pad. Always 000
  head[0] = 0.0f;
  head[1] = 1.0f;
  head[2] = 0.0f;
// Determine foot roll
  dotp = PFDOT_VEC3(head, foot_data[leg].normal);
  pfMakeRotMat(roll, (pfArcCos(dotp) - 90.0f), 0.0f, 1.0f, 0.0f);

// Set heading of foot pad + 90 degrees
  head90[0] = 1.0f;
  head90[1] = 0.0f;
  head90[2] = 0.0f;
// Determine foot pitch
  dotp = PFDOT_VEC3(head90, foot_data[leg].normal);
  pfMakeRotMat(pitch, (90.0f - pfArcCos(dotp)), 1.0f, 0.0f, 0.0f);
  pfPostMultMat(pitch, roll);
  pfPreMultMat(foot_data[leg].foot_mat, pitch);

// Set the foot pad DCS to the calculated orientation
  pfDCSMatrix(DCSlink[leg][3], foot_data[leg].foot_mat);

 } // end for statement
} // end set_initial_foot_position
```

```
// This function establishes the location and direction of the intersection segments being
// projected from each foot.

void set_foot_intersection_segments(pfSeg **segment, int leg) {

 static pfVec3 foot_segs_xyz[3][RAYS_PER_SECTOR];
 static float angle = 120.0f / (RAYS_PER_SECTOR - 1);
 static int init = 1;
 int count = 0;
 float s, c;
 pfVec3 temp_vec1, temp_vec2;


 if(init) {
  for (int sect = 0; sect<3; sect++) {
   pfSinCos(120.0f*sect+60.0, &s, &c);
   PFSET_VEC3(foot_segs_xyz[sect][0], -s*3.0f, c*3.0f, 0.0f);
   segment[count++] = (pfSeg*)pfMalloc(sizeof(pfSeg), NULL);

   for (int nseg = 0; nseg<RAYS_PER_SECTOR - 1; nseg++) {
    pfSinCos((angle*nseg)+(angle/2.0f)+(120.0f*sect), &s, &c);
    PFSET_VEC3(foot_segs_xyz[sect][nseg+1], -s*FOOTPAD_RADIUS, c*FOOTPAD_RADIUS, 0.0f);
    segment[count++] = (pfSeg*)pfMalloc(sizeof(pfSeg), NULL);
   }
  }
  init = 0;
  count = 0;
 }

 temp_vec1[0] = foot_data[leg].foot_xyz[2];
 temp_vec1[1] = foot_data[leg].foot_xyz[0];
 temp_vec1[2] = foot_data[leg].foot_xyz[1] + 5.0f;

 /* make several rays looking "down" at terrain */

 for (int sect = 0; sect<3; sect++) {
  for (int nseg = 0; nseg<RAYS_PER_SECTOR; nseg++) {

   PFADD_VEC3(segment[count]->pos, temp_vec1, foot_segs_xyz[sect][nseg]);
   PFSET_VEC3(segment[count]->dir, 0.0f, 0.0f, -1.0f);
   pfNormalizeVec3(segment[count]->dir);
   segment[count]->length = 20.0f;
   count++;
  }
 }

} // end set_foot_intersection_segments
```

```
// This function determines the pfPlane on which to orient the foot pads

void orient_foot(pfIsect data[3*RAYS_PER_SECTOR+1], int leg) {

  pfVec3 contact_points[3];
  static pfPlane *ground = (pfPlane*)pfMalloc(sizeof(pfPlane), NULL);
  int count = 0; // Keeps track of the number of segments that needs
          // to be considered

  pfVec3 head[3][RAYS_PER_SECTOR], head90;
  float dotp;
  float delta_height[3][RAYS_PER_SECTOR];
  float deepest = 0.0f;
  float sh, ch;
  int   sector = 0;
  int   seg_ray = 0;
  pfMatrix m2, m3;

  static float angle = 120.0f / (RAYS_PER_SECTOR - 1);
  static pfSeg  *foot = (pfSeg*)pfMalloc(sizeof(pfSeg),NULL);

  for (int sect = 0; sect<3; sect++) {
   PFSET_VEC3(contact_points[sect], 0.0f, 0.0f, 0.0f);

   for (int nseg = 0; nseg<RAYS_PER_SECTOR; nseg++) {
    if (data[count].flags & (PFIS_POINT|PFIS_NORM|PFIS_PRIM)) {  // Data is good
     pfXformPt3(data[count].point, data[count].point, data[count].xform);

     if(data[count].point[2] > contact_points[sect][2])
      PFCOPY_VEC3(contact_points[sect], data[count].point);
    }
    count++;
   }
  }

  PFSET_VEC3(foot->dir, 0.0f, 0.0f, -1.0f);
  foot->length = 50.0f;
  foot->pos[0] = foot_data[leg].foot_xyz[2];
  foot->pos[1] = foot_data[leg].foot_xyz[0];
  foot->pos[2] = foot_data[leg].foot_xyz[1];

  pfMakePtsPlane(ground, contact_points[2], contact_points[1], contact_points[0]);
  pfNormalizeVec3(ground->normal);

  if (ground->normal[2] < 0.0f) {
   pfMakePtsPlane(ground, contact_points[0], contact_points[1], contact_points[2]);
   pfNormalizeVec3(ground->normal);
  }

  count = 0;

  if(pfSegIsectPlane(foot, ground, &foot_data[leg].foot_height) == PFIS_FALSE)
    printf("No ground intersection 1 for leg %i\n", leg);

  head[0][0][0] = 0.0f;
  head[0][0][1] = 1.0f;
  head[0][0][2] = 0.0f;
  dotp = PFDOT_VEC3(head[0][0], ground->normal);
  pfMakeRotMat(m2, (pfArcCos(dotp) - 90.0f), 1.0f, 0.0f, 0.0f);
  head90[0] = 1.0f;
  head90[1] = 0.0f;
```

```
head90[2] = 0.0f;
dotp = PFDOT_VEC3(head90, ground->normal);

pfMakeRotMat(m3, (90.0f - pfArcCos(dotp)), 0.0f, 1.0f, 0.0f);
pfPreMultMat(m3, m2);

for (sect = 0; sect<3; sect++) {

 pfSinCos(120.0f*sect+60.0, &sh, &ch);
 PFSET_VEC3(head[sect][0], -sh, ch, 0.0f);
 pfXformPt3 (head[sect][0], head[sect][0], m3);
 pfNormalizeVec3(head[sect][0]);
 pfScaleVec3(head[sect][0], 3.0f, head[sect][0]);
 delta_height[sect][0] = foot_data[leg].foot_xyz[1] -
                 foot_data[leg].foot_height +
                 head[sect][0][2] - data[count++].point[2];

 if (delta_height[sect][0] < deepest) {
  deepest = delta_height[sect][0];
  sector  = sect;
  seg_ray = count - 1;
 }

 for (int nseg = 0; nseg<RAYS_PER_SECTOR - 1; nseg++) {
  pfSinCos((angle*nseg)+(angle/2.0f)+(120.0f*sect), &sh, &ch);
  PFSET_VEC3(head[sect][nseg+1], -sh, ch, 0.0f);
  pfXformPt3 (head[sect][nseg+1], head[sect][nseg+1], m3);
  pfNormalizeVec3(head[sect][nseg+1]);
  pfScaleVec3(head[sect][nseg+1], FOOTPAD_RADIUS, head[sect][nseg+1]);
  delta_height[sect][nseg+1] =  foot_data[leg].foot_xyz[1] -
                    foot_data[leg].foot_height +
                    head[sect][nseg+1][2] - data[count++].point[2];

  if (delta_height[sect][nseg+1] < deepest) {
   deepest = delta_height[sect][nseg+1];
   sector  = sect;
   seg_ray = count - 1;
  }
 }

 if (deepest <= -2.5f) {
  PFCOPY_VEC3(contact_points[sector], data[seg_ray].point);
  deepest = 0.0f;
 }
}

pfMakePtsPlane(ground, contact_points[2], contact_points[1], contact_points[0]);
pfNormalizeVec3(ground->normal);

if (ground->normal[2] < 0.0f) {
 pfMakePtsPlane(ground, contact_points[0], contact_points[1], contact_points[2]);
 pfNormalizeVec3(ground->normal);
}

if(pfSegIsectPlane(foot, ground, &foot_data[leg].foot_height) == PFIS_FALSE)
  printf("No ground intersection 2 for leg %i\n", leg);

PFCOPY_VEC3(foot_data[leg].normal, ground->normal);
foot_data[leg].foot_xyz[1] -= (foot_data[leg].foot_height - LINK4LENGTH);
foot_data[leg].ground_contact = TRUE;
} // end orient_foot
```

```
void move_camera_boom(pfDCS *DCSboom[3]) {


 pfDCSRot(DCSboom[0], (float)getvaluator(DIAL0), 0.0f, 0.0f);
 pfDCSRot(DCSboom[1], -(float)getvaluator(DIAL1),0.0f, 0.0f);
 pfDCSRot(DCSboom[2], -(float)getvaluator(DIAL2),0.0f, 0.0f);
}



// Read into memory materials and textures used in the simulation
void read_materials() {

   if (LoadGDL2("models/materials.gdl") == NULL)
      fprintf(stderr,"Unable to initialize materials\n");

   if (LoadGDL2("models/showgdl.textures.gdl") == NULL)
      fprintf(stderr,"Unable to initialize textures\n");

} // end read_materials



// Read in the terrain model and assign unique characteristics
pfGroup* read_terrain() {

 pfGroup *G_dirt, *pond;

 G_dirt = LoadGDL2("models/test_floor.gdl");

 pond = LoadGDL2("models/pond.gdl");

// Set up the intersection mask for the terrain root node
 pfNodeTravMask(G_dirt,PFTRAV_ISECT,
            TERRAIN_MASK, PFTRAV_SELF|PFTRAV_DESCEND|PFTRAV_IS_CACHE, PF_SET);

 pfNodeTravMask(pond,PFTRAV_ISECT,
            WATER_MASK, PFTRAV_SELF|PFTRAV_DESCEND|PFTRAV_IS_CACHE, PF_SET);

 pfAddChild (G_dirt, pond);

 return(G_dirt);

} // end read_terrain
```

```
// ***************************************************************************
// FILENAME:   walk.C
// PURPOSE:    This file contains the functions to perform a simplistic tripod gait
//
// NOTE:       This program is written in C++ utilizing Performer 1.0
// AUTHOR:     Wrus Kristiansen and John Goetz
// DATE:       20 February 1994
// COMMENT:
// UPDATE:
// ***************************************************************************
/
// External access to two functions: (prototypes in "robot_globals.h")
//
//   1. void restart_robot();
//      Initializes all robot parameters,
//      placing the robot at the origin at rest.
//
//   2. void move_robot(fcs joystick);
//      Reads the system clock and updates robot position.
//      Joystick is accessed for an "ordered velocity" at the
//      beginning of each step. (Joystick port is assumed
//      to have been successfully opened prior to call.)
//      Robot body accelleration is non-infinite.
//


#include <iostream.h>      // for error messages
#include <stdio.h>
#include <stdlib.h>        // for exit stmt
#include <math.h>          // for trig functions and sqrt
#include <sys/times.h>     // for retrieving system time
#include <sys/param.h>
#include "pf.h"

// Joystick attached to HOTAS Flight Control System is
// used for desired velocity input for the robot.
// "fcs.H" and "fcs.C", written by Paul Barham, are used
// for access to the device.
#include "fcs.H"           // for joystick access


// aqua robot definitions and globals passed to draw function
#define __COMMON__
#include "robot_globals.h"

// LOCAL DECLARATIONS

// joystick stuff

// minimum non-zero value recognized for joystick input
// (smaller values are zeroed)
#define JOY_MIN 0.1

// Name for joystick port
// gravy3&5 portname
#define JOY_STICK_PORT "/dev/ttyd4"

// instanciate fcs object
char *port_name = JOY_STICK_PORT;
fcs joystick (port_name, 0, 0);
// end of joystick stuff
```

```
// for angle conversions (radians to/from degrees)
#define PI 3.1416f
#define RADTODEG 57.2958f
#define DEGTORAD 0.0174533f
#define MINIMUM(a,b) ((a<b)? a:b)
#define HYPT(a,b) pfSqrt((a*a)+(b*b))
#define LINK2SQR (LINK2LENGTH * LINK2LENGTH)
#define LINK3SQR (LINK3LENGTH * LINK3LENGTH)


float delta_time;          // time since last update
float scalar_speed;        // for robot body
float joint_angles[18];


pfVec3 tp1_pos;            // XYZ center of legs 1,3,5 (tripod 1)
pfVec3 tp2_pos;            // XYZ center of legs 2,4,6 (tripod 2)

int  moving_leg_group;     // IDs stepping tripod (0=none)
int  step_in_progress;     // boolean
int  reset;    // boolean to flag a reset

// Center of body
pfVec3 robot_position;

// planned end of step positions
pfVec3 moving_group_goal_pos; // tripod position
pfVec3 body_goal_pos;      // robot body position

pfVec3 actual_velocity;

float body_distance_this_step;// scalar distances to determine
float distance_to_go;      // pct of step completed

// LOCAL PROTOTYPES

// Determines which tripod will take the next step.
// Calculates and saves goal positions for body and stepping tripod.
void  plan_motion(pfVec3 ordered);

// Increments stepping tripod position for new frame.
void  move_tripod_group(int end_of_step);


// Calculates leg joint angles for given
// robot body position and tripod positions.
void  calculate_joint_angles();

void move_performer_DCS(pfDCS *base, pfDCS *DCSlink[6][4]);

// Reads joystick of an FCS object and returns the horizontal
// component in X (left -1 to right +1) and the vertical component
// in Z (ahead -1 to back +1). The vector sum of the two components
// is normalized if greater than one and zeroed if less than JOYMIN.
void  read_joystick(pfVec3 *ordered);
```

```
// Establish the initial posture for Aquarobot
void restart_robot(pfDCS *base, pfDCS *DCSlink[6][4]) {

 static pfVec3 init_robot_position;

// Place robot in air so it can be placed on terrain correctly.
 pfSetVec3(init_robot_position, 50.0f, ROBOT_HEIGHT, 0.0f);

 // joystick setup
 // Check to make sure the FCS was found on the desired port
 // and that the FCS is communicating correctly.

 if (!joystick.exists()) { // abort program
    cerr << "Unable to communicate with joystick." << endl;
    exit(0); // quit
 }

 else { // run program

    // Deaden the joystick pitch and roll so user can rest
    // hand on stick without causing movement.
    joystick.deaden_pitch(0.05);
    joystick.deaden_roll(0.05);

    // Clear joystick input
    // (needed for reset so robot doesn't see old value)
    if (joystick.new_data())
    {
      joystick.get_data();
      joystick.clear_data();
    }
 }
 // end joystick setup

 // initialize local variables
 pfInitClock();
 scalar_speed = 0.0f;

 moving_leg_group = 0;          // initially
 step_in_progress = 0;          // initially false
 reset = 1;                     // initially true

 // initialize robot's foot positions
 pfSetVec3(moving_group_goal_pos, init_robot_position[0], LINK4LENGTH,
        init_robot_position[2]);

 pfSetVec3(tp1_pos, init_robot_position[0], 15.0f,
        init_robot_position[2]);

 pfSetVec3(tp2_pos, init_robot_position[0], 15.0f,
        init_robot_position[2]);

 // initialize robot's body position
 pfCopyVec3(body_goal_pos, init_robot_position);
 pfCopyVec3(robot_position, init_robot_position);

 for (int leg = 0; leg<6; leg++) {
  foot_data[leg].ground_contact = TRUE;
  foot_data[leg].direction = FOOT_UP;
  foot_data[leg].foot_xyz[1] = 0.0f;
 }
```

```
  // initialize robot's leg joints
  calculate_joint_angles();

  move_performer_DCS(base, DCSlink);

  tp1_pos[1] = LINK4LENGTH;
  tp2_pos[1] = LINK4LENGTH;
}



// Determine location of center of body based on joystick input

void move_robot(pfDCS *base, pfDCS *DCSlink[6][4]) {

  pfVec3 temp;
  pfVec3 ordered_velocity;

  static float old_time;
  float current_time = pfGetTime();
  int end_of_step = 0;  // boolean used to detect end of step, signal to put feet down.

  if (reset)
    old_time = 0.0f;

  delta_time = current_time - old_time;
  old_time   = current_time;
  if (delta_time > 1.0f)
    delta_time = 0.05f;

  if (!step_in_progress) { // If no step is currently in progress

    // get ordered velocity components
    read_joystick(&ordered_velocity);
    PFSCALE_VEC3(ordered_velocity, MAX_SPEED, ordered_velocity);
    PFCOPY_VEC3(actual_velocity, ordered_velocity);

    // save scalar value
    scalar_speed = PFLENGTH_VEC3(ordered_velocity);

    // plan the step
    plan_motion(ordered_velocity);
  }

  // calculate distance remaining on this step
  PFSUB_VEC3(temp, body_goal_pos, robot_position);
  distance_to_go = PFLENGTH_VEC3(temp);



  if (distance_to_go >= delta_time * 10.0f)  //prevent over-shooting goal
    PFSCALE_VEC3(actual_velocity, (scalar_speed/distance_to_go), temp);

  else
    end_of_step = 1;

  // update robot's body position
  PFSCALE_VEC3(temp, delta_time, actual_velocity);
  PFADD_VEC3(robot_position, robot_position, temp);
```

```
  // update stepping tripod position
  if (step_in_progress)
    move_tripod_group(end_of_step); // move the stepping tripod group

  calculate_joint_angles(); // update all eighteen joints

  move_performer_DCS(base, DCSlink); // Set Performer DCS's

  reset = 0;
}

// Set bodies goal position based on joystick input

void  plan_motion(pfVec3 ordered) {

  pfVec3 temp;

  // detect end of motion condition
  if (scalar_speed == 0.0f) // stop ordered
  {
    // and was previously walking (tp1_pos not= tp2_pos)
    if (!PFEQUAL_VEC3(tp1_pos, tp2_pos)) {

      // plan next step to go to rest position
      switch (moving_leg_group)
      {
        // alternate groups
        case (1):
          // set goals for rest position

          pfCopyVec3(moving_group_goal_pos, tp1_pos);
          pfCopyVec3(body_goal_pos, tp1_pos);
          body_goal_pos[1] = ROBOT_HEIGHT;

          // switch groups
          moving_leg_group = 2;

          // enable step
          step_in_progress = 1;
          break;

        case (2):
          // set goals for rest position

          pfCopyVec3(moving_group_goal_pos, tp2_pos);
          pfCopyVec3(body_goal_pos, tp2_pos);
          body_goal_pos[1] = ROBOT_HEIGHT;

          // switch groups
          moving_leg_group = 1;

          // enable step
          step_in_progress = 1;
          break;

      } // end switch

      // set half speed
      scalar_speed = 0.5f * MAX_SPEED;
    }
```

```
  else
  {
    // already in rest position (still in "stop ordered" block)
    moving_leg_group = 0;
  }// end if
}


else // scalar_speed (ordered speed) > 0
{
  // plan next step.
  switch (moving_leg_group)
  {
    // alternate groups
    case (1):
      // switch groups
      moving_leg_group = 2;
      // enable step
      step_in_progress = 1;

      // determine goal position for moving leg group

      pfSetVec3(moving_group_goal_pos, tp1_pos[0] + STRIDE_TIME * ordered[0], LINK4LENGTH,
            tp1_pos[2] + STRIDE_TIME * ordered[2]);

      // determine goal position for body
      pfSetVec3(body_goal_pos, tp1_pos[0] + (STRIDE_TIME * ordered[0] / 2.0f),
            ROBOT_HEIGHT, tp1_pos[2] + (STRIDE_TIME * ordered[2] / 2.0f));

      break;
                        // Y positions are a constant

    case (0):
    case (2):
      // switch groups
      moving_leg_group = 1;
      // enable step
      step_in_progress = 1;

      // determine goal position for moving leg group
      pfSetVec3(moving_group_goal_pos, tp2_pos[0] + STRIDE_TIME * ordered[0], LINK4LENGTH,
            tp2_pos[2] + STRIDE_TIME * ordered[2]);

      // determine goal position for body
      pfSetVec3(body_goal_pos, tp2_pos[0] + (STRIDE_TIME * ordered[0] / 2.0f),
            ROBOT_HEIGHT, tp2_pos[2] + (STRIDE_TIME * ordered[2] / 2.0f));
      break;

  } // end switch

} // end if (scalar_speed == 0)

if (step_in_progress) // new step started
{
  // set length of step for pct completion calculation
  body_distance_this_step = PFDISTANCE_PT3(body_goal_pos, robot_position);
}// end if

} // end plan_motion
```

```
void move_tripod_group(int end_of_step) {

  float s, c; // Places for sin and cos
  float foot_lift;
  pfVec3 temp;
  static float last_tp1_height = 0.0f, last_tp2_height = 0.0f;

  switch (moving_leg_group)
  {
    case (1):
      if (end_of_step)
      {
        // snap position to end of step
        pfSetVec3(tp1_pos, moving_group_goal_pos[0],
              LINK4LENGTH, moving_group_goal_pos[2]);

        // reset step_in_progress
        step_in_progress = 0;

        if(!foot_data[0].ground_contact) {
          foot_data[0].ground_contact = TRUE;
          PFSET_VEC3(foot_data[0].normal, 0.0f, 0.0f, 1.0f);
        }
        if(!foot_data[2].ground_contact) {
          foot_data[2].ground_contact = TRUE;
          PFSET_VEC3(foot_data[2].normal, 0.0f, 0.0f, 1.0f);
        }
        if(!foot_data[4].ground_contact) {
          foot_data[4].ground_contact = TRUE;
          PFSET_VEC3(foot_data[4].normal, 0.0f, 0.0f, 1.0f);
        }
        foot_data[0].direction = FOOT_UP;
        foot_data[2].direction = FOOT_UP;
        foot_data[4].direction = FOOT_UP;

        last_tp1_height = 0.0f;
      }
      else
      {
        if (distance_to_go <= body_distance_this_step) // no overshoot
        // required in case body momentum is away from
        // ordered direction of motion. Delays picking
        // up feet for next step until stabilazation.
        {
          pfSinCos((RADTODEG * (PI * distance_to_go / body_distance_this_step)), &s, &c);
          foot_lift = body_distance_this_step * s;

          PFSCALE_VEC3(temp, (2.0f * delta_time), actual_velocity);
          tp1_pos[1] = LINK4LENGTH + MINIMUM(MAX_FOOT_LIFT,foot_lift);

          if ((last_tp1_height > tp1_pos[1]) &&
            (!foot_data[0].direction)) {
              foot_data[0].direction = FOOT_DOWN;
              foot_data[2].direction = FOOT_DOWN;
              foot_data[4].direction = FOOT_DOWN;
          }

          last_tp1_height = tp1_pos[1];
          PFADD_VEC3(tp1_pos, tp1_pos, temp);

        }// end if
```

```
      }// end if
      break;
    // end case (1)

    case (2):
      if (end_of_step)
      {
        // snap position to end of step
        pfSetVec3(tp2_pos, moving_group_goal_pos[0],
              LINK4LENGTH, moving_group_goal_pos[2]);

        // reset step_in_progress
        step_in_progress = 0;

        if(!foot_data[1].ground_contact) {
          foot_data[1].ground_contact = TRUE;
          PFSET_VEC3(foot_data[1].normal, 0.0f, 0.0f, 1.0f);
        }
        if(!foot_data[3].ground_contact) {
          foot_data[3].ground_contact = TRUE;
          PFSET_VEC3(foot_data[3].normal, 0.0f, 0.0f, 1.0f);
        }
        if(!foot_data[5].ground_contact) {
          foot_data[5].ground_contact = TRUE;
          PFSET_VEC3(foot_data[5].normal, 0.0f, 0.0f, 1.0f);
        }
        foot_data[1].direction = FOOT_UP;
        foot_data[3].direction = FOOT_UP;
        foot_data[5].direction = FOOT_UP;

        last_tp2_height = 0.0f;
      }
      else
      {
        if (distance_to_go <= body_distance_this_step) // no overshoot
        {
          pfSinCos((RADTODEG * (PI * distance_to_go / body_distance_this_step)), &s, &c);
          foot_lift = body_distance_this_step * s;

          PFSCALE_VEC3(temp, (2.0f * delta_time), actual_velocity);
          tp2_pos[1] = LINK4LENGTH + MINIMUM(MAX_FOOT_LIFT,foot_lift);

          if ((last_tp2_height > tp2_pos[1]) &&
            (!foot_data[1].direction)) {
              foot_data[1].direction = FOOT_DOWN;
              foot_data[3].direction = FOOT_DOWN;
              foot_data[5].direction = FOOT_DOWN;
          }

          last_tp2_height = tp2_pos[1];
          PFADD_VEC3(tp2_pos, tp2_pos, temp);
        }// end if
      }// end if
      break;
    // end case (2)
  }// end switch
}// end move_tripod_group

void calculate_joint_angles() {

  pfVec3 joint1_pos; // XYZ coordinates
```

```
pfVec3 joint2_pos; // "     "
pfVec3 temp;
float m1, m2, r, dy, dxz, rsqr;
float s, c;  // Holds sin/cos values
register int leg;          // loop variable (0 = leg 1)

for(leg = 0; leg < 6; leg++) {

  // determine joint 1 position
  PFSET_VEC3(temp, offset_factor[leg*2]*LINK0LENGTH,
        0.0f, offset_factor[leg*2+1]*LINK0LENGTH);
  PFADD_VEC3(joint1_pos, robot_position, temp);


   // determine foot position
   switch (leg) {
    case (0): case (2): case (4): // legs 1,3&5 on tripod 1

      if ((!foot_data[leg].ground_contact) ||
        ((!foot_data[leg].direction) &&
         (tp1_pos[1] >= foot_data[leg].foot_xyz[1]))) {

          PFSET_VEC3(temp, (offset_factor[leg*2]*FOOT_RADIUS),
             0.0f, (offset_factor[leg*2+1]*FOOT_RADIUS));

          PFADD_VEC3(foot_data[leg].foot_xyz, tp1_pos, temp);
          if (step_in_progress)
            foot_data[leg].ground_contact = FALSE;
       }
       break;

    case (1): case (3): case (5): // legs 2,4&6 on tripod 2

      if ((!foot_data[leg].ground_contact) ||
        ((!foot_data[leg].direction) &&
         (tp2_pos[1] >= foot_data[leg].foot_xyz[1]))) {

          PFSET_VEC3(temp, (offset_factor[leg*2]*FOOT_RADIUS),
             0.0f, (offset_factor[leg*2+1]*FOOT_RADIUS));

          PFADD_VEC3(foot_data[leg].foot_xyz, tp2_pos, temp);
          if (step_in_progress)
            foot_data[leg].ground_contact = FALSE;
      }
      break;
  } // end switch


// calculate joint 1 angle using slopes
   // slope of line from body center to joint 1
   m1 =  (joint1_pos[2] - robot_position[2])  // dZ
          /(joint1_pos[0] - robot_position[0]); // dX
   // slope of line from joint 1 to position directly above foot
   m2 =  (foot_data[leg].foot_xyz[2] - joint1_pos[2])  // dZ
          /(foot_data[leg].foot_xyz[0] - joint1_pos[0]); // dX
joint_angles[3*leg] = pfArcTan2((m2 - m1), (1.0f + (m1 * m2)));

// determine joint 2 position

pfSinCos(RADTODEG * ((PI*leg/3.0f) + (DEGTORAD * joint_angles[3*leg])), &s, &c);
```

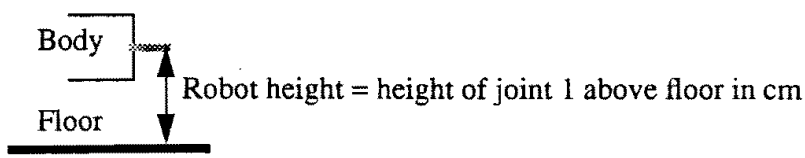
```
    PFSET_VEC3(temp, LINK1LENGTH * c, 0.0f, LINK1LENGTH * s);

    PFADD_VEC3(joint2_pos, joint1_pos, temp);

    // determine length of
    // the third side of triangle (joint 2, joint 3, foot).
    dy = joint1_pos[1] - foot_data[leg].foot_xyz[1];          // Y coordinate

    dxz = HYPT((joint2_pos[0] - foot_data[leg].foot_xyz[0]), // X coordinate
                (joint2_pos[2] - foot_data[leg].foot_xyz[2]));// Z coordinate
    r = HYPT(dy, dxz); // length of third side

    // need squares of all three sides for "law of cosines"
    rsqr = r * r;

    // calculate joint2 angle using law of cosines
    joint_angles[3*leg+1] =
      pfArcCos((LINK2SQR + rsqr - LINK3SQR) // A3 (angle opposite
          /(2.0f * LINK2LENGTH * r))     //    LINK3LENGTH)
      - pfArcSin(dy / r);                 // - phi

    // calculate joint3 angle using law of cosines
    joint_angles[3*leg+2] =           // Ar (angle opposite r)
      pfArcCos((LINK2SQR + LINK3SQR - rsqr)
          /(2.0f * LINK2LENGTH * LINK3LENGTH));
  } // end for loop


} // end calculate_joint_angles
```

```
// Update DCS's to display new position
void move_performer_DCS(pfDCS *base, pfDCS *DCSlink[6][4]) {

  static pfMatrix m1;
// Move robots center of body (convet to robotics coordinate system)
  pfDCSTrans(base, robot_position[0], robot_position[2], -robot_position[1]);


  for (int leg = 0; leg < 6; leg++) {

    pfDCSRot(DCSlink[leg][0], joint_angles[3*leg], 0.0f, 0.0f);
    pfDCSRot(DCSlink[leg][1], joint_angles[3*leg+1], 0.0f, 0.0f);
    pfDCSRot(DCSlink[leg][2], 180.0f + joint_angles[3*leg+2], 0.0f, 0.0f);

// Translate foot into correct position (world coordinates)
    pfMakeTransMat(foot_data[leg].foot_mat, foot_data[leg].foot_xyz[0],
          foot_data[leg].foot_xyz[2], -foot_data[leg].foot_xyz[1]);

// If foot is in the air, keep it level and have it rotate with shoulder
    if (!foot_data[leg].ground_contact) {
      pfMakeRotMat(m1, joint_angles[3*leg], 0.0f, 0.0f, 1.0f);
      pfPostMultMat(m1, foot_data[leg].foot_mat);
      pfDCSMatrix(DCSlink[leg][3], m1);
    }

  } // end for statement

} // end move_performer_DCS




// Check the joystick for response
void  read_joystick(pfVec3 *ordered) {

  pfVec3 stick_direction;

  if (joystick.new_data()) {

    // read joystick data
    joystick.get_data();

    pfSetVec3(stick_direction, joystick.roll(), 0.0f, joystick.pitch());
    pfNormalizeVec3(stick_direction);

    // mark current data as used
    joystick.clear_data();

    if (pfLengthVec3(stick_direction) < JOY_MIN)
      pfSetVec3(stick_direction, joystick.roll(), 0.0f, joystick.pitch());
  } // end if

  // write return values
  PFCOPY_VEC3(*ordered, stick_direction);
} // end read_joystick
```

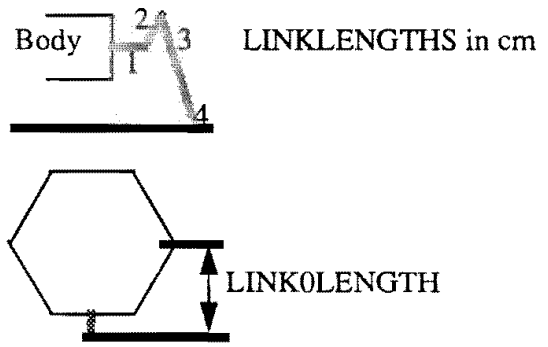
APPENDIX B: SIMULATION SUPPORT CODE

```
//
// robot_globals.h
//
// written by Wrus Kristiansen and John Goetz
//
// Global definitions, variables and animation function prototypes
// for aquarobot, a six legged underwater walking robot.
//
// Aquarobot's positional data are calculated in functions
// and passed through variables declared in this header to
// external robot drawing function written by John Goetz.
//
// Function definitions are in "walk.C".
//


#ifndef __ROBOT_GLOBALS_H
#define __ROBOT_GLOBALS_H

#include "pf.h"

#ifdef __COMMON__
#define EXTERN
#else
#define EXTERN extern
#endif

typedef struct {

  pfVec3 foot_xyz;
  pfMatrix foot_mat;
  pfVec3 normal;
  int ground_contact;
  int direction;
  float foot_height;

} FOOT_DATA;

EXTERN
FOOT_DATA foot_data[6];

// Defines for foot direction
#define FOOT_UP 0
#define FOOT_DOWN 1

//////////////////////////////////////////////////////
//                                                  //
// all positional information is based on a left handed   //
// XYZ coordinate system: +X = right, +Y = up, +Z = toward  //
// viewer (out from screen). (SCALE 1.0 = 1 centimeter)    //
//                                                  //
//////////////////////////////////////////////////////
```

```
//
// DEFINITIONS
//
// Any of these may be adjusted; however, care must be taken
// not to exceed physical limitations. For example, the product
// of MAX_SPEED and STRIDE_TIME should not exceed 50 for the
// current dimentional limitations of the robot. i.e. it can't
// take 2 second steps, going 400cm/sec because its legs can't
// reach far enough to move the body 8 meters per step...
//
// maximum robot speed in cm/sec
#define MAX_SPEED  33.33f   //25.0f

// radius of feet positions in cm
#define FOOT_RADIUS 109.0f

// max height foot lifted during step
#define MAX_FOOT_LIFT 20.0f  //15.0f

// time, in seconds, for one step
#define STRIDE_TIME  1.5f  //2.0f

// The floor for the robot to walk on is the plane
// described by Y = 0.0.

#define FLOOR_LEVEL 0.0f
```

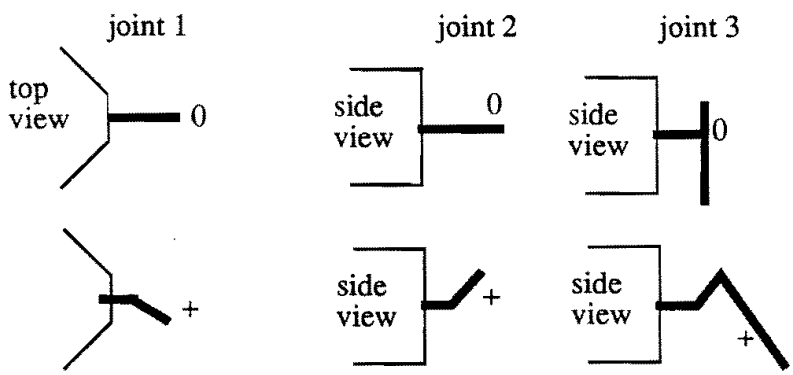



```
#define ROBOT_HEIGHT 102.0f   // 92.0f

//
// ROBOT DIMENSIONS
//
```




```
/* link lengths */
#define LINK0LENGTH 37.5f
#define LINK1LENGTH 20.0f
#define LINK2LENGTH 52.0f
#define LINK3LENGTH 102.0f
#define LINK4LENGTH 3.0f
```

```
//
// Offset matrix.
//
// For position of first joint of a leg, multiply X and Z
// components by bodyradius and add to body position;
// For position of a foot, multiply X and Z components
// by FOOT_RADIUS and add to appropriate tripod position.
// Leg 1 is on +X axis, and legs 2 through 6 are
// at 60 degree intervals.

const float offset_factor[12] = {
   //       Xcomponent  Zcomponent
   // leg 1   cos(0)      sin(0)
              1.0,       0.0,
   // leg 2   cos(60)     sin(60)
              0.5,       0.866,
   // leg 3   cos(120)    sin(120)
             -0.5,       0.866,
   // leg 4   cos(180)    sin(180)
             -1.0,       0.0,
   // leg 5   cos(240)    sin(240)
             -0.5,       -0.866,
   // leg 6   cos(300)    sin(300)
              0.5,       -0.866};

// aqua robot's body position. (X, Y, Z coordinates)
//
// Since orientation is constant, this is sufficient
// to totally describe the postition of the body.
//
// aqua robot's leg joint angles.
//
// Three joint angles are defined for each leg.
// The joints and their 0, + and - angles are
// pictorially described as follows:
//
//
```




```
// current angles for all joints
// leg 1, joint 1, 2, 3,
// leg 2, joint 1, 2, 3, ...
// leg 6, joint 1, 2, 3,
```

```
// FUNCTION PROTOTYPES
//

// Initializes all robot global variables, positioning
// the robot at the origin in the rest position.
void restart_robot(pfDCS *base, pfDCS *DCSlink[6][4]);

// Reads system clock and updates robot position
// and joint angles.
// Reads fcs object, joystick pitch and roll components,
// for motion orders. FCS object must be valid!
// Returns robot to rest position, body and tripod
// centers having equal X and Z coordinates, when
// ordered velocity is 0.
void move_robot(pfDCS *base, pfDCS *DCSlink[6][4]);


#endif  // __ROBOT_GLOBALS_H
```

```
// ********************************************************************************
// FILENAME: SGIwindowcam.C
// PURPOSE:  This program contains the functions for queing devices and
//           checking the events queue for mouse and spaceball events. It
//           also contains the functions for controling the camera and focal
//           point.
//
// NOTE:     This is and IRIS 3D program written in C++
// AUTHOR:   John Goetz,
// DATE:     15 January 1994
// COMMENT:
// UPDATE:
// ********************************************************************************

#include <gl/gl.h>
#include <stdlib.h>
#include <gl/device.h>
#include <gl/spaceball.h>
#include "SGIwindowcam.H"
#include "pf.h"

#define RESET 14  // pop up menu item
#define EXIT  15  // pop up menu item

#define RADTODEG  57.29577951f
#define SBRATIO  0.0004f   // spaceball rotational scale factor
#define SBTRATIO  0.008f   // spaceball translational scale factor
#define PIE      3.141592654f
#define PIEOVER2  1.570796327f
#define TWOPIE    6.283185308f

long topmenu; // pop up menu hook


/*****************************************************************************/
/* Function initialize                                                       */
/*****************************************************************************/
void initialize() {


// Que the devices and keys to be used
   qdevice(REDRAW);        // queue the redraw device
   qdevice(MENUBUTTON);    // queue the menubutton
   qdevice(LEFTMOUSE);     // Initialize Mouse Buttons to be Queued
   qdevice(SBTX);    // SpaceBall Translate in X
   qdevice(SBTY);
   qdevice(SBTZ);
   qdevice(SBRX);    // SpaceBall Rotate about X
   qdevice(SBRY);
   qdevice(SBRZ);
   qdevice(SBPERIOD);    // SpaceBall time delta
   qdevice(ESCKEY);    // Exits program
   qdevice(F1KEY);    // Toggles statistical information
   qdevice(F2KEY);    // Toggles fog

// Set the limits for the dials to be used
   setvaluator (DIAL0, 0, -10000, 10000);
   setvaluator (DIAL1, 0, -40, 50);
   setvaluator (DIAL2, 0, -90, 90);
```

```
// Set dead band for the dials to be used
   noise(DIAL0, 2);
   noise(DIAL1, 2);
   noise(DIAL2, 2);

// Create the pop-up menus to reset simulation and exit program
   makethemenus();

} // end initialize()


/*****************************************************************************/
/* Function Make_the_Menus                                                   */
/*****************************************************************************/
void makethemenus() {

  // build the top level pop-up menu
    topmenu = defpup("Reset %x14| Exit %x15");
}


/*****************************************************************************/
/* Function Check_Que                                                        */
/*****************************************************************************/
void Check_Que(SharedData *data) {

   long device;
   long  hititem;  // variable holding hit name
   short value;    // value returned from the event queue
   short sbvals[7]; // array to hold spaceball values
   short kbvals[7]; // array to hold keyboard values

   // do we have something on the event queue?
   while (qtest())
   {
  device = qread(&value);
switch (device)
{
 // Redraw window after resizing, not needed for a fixed windowsize
 case REDRAW:
   reshapeviewport();
   break;

 case MENUBUTTON:     // Menu selections
   if (value == 1)
   {
     /* which popup selection do we want? */
     hititem = dopup(topmenu);

           switch (hititem) {
             case RESET:
       data->resetFlag = 1;
       break;

      case EXIT:
       data->exitFlag = 1;
       break;
     }

   } // end if for MENUBUTTON
   break;
```

```
case (SBPERIOD):
case (SBTX):
case (SBTY):
case (SBTZ):
case (SBRX):
case (SBRY):
case (SBRZ):
  sbvals[device-SBTX] = value;
  if (device == SBRZ)
    calculate_view(sbvals, &data->view);
  break;

/* ESC-key signals end of simulation */
case ESCKEY:
    data->exitFlag = 1;
    break;

/* F1-key toggles channel-profile display */
case F1KEY:
  if (value == 1)
    data->pfStats = !data->pfStats;
  break;

/* F2-key toggles underwater environment */
case F2KEY:
  if (value == 1)
    data->pfFog = !data->pfFog;
  break;

default:
     break;
} // end switch
  } // end while qtest
  qreset();
} // end of Check_Que
```

```
/*****************************************************************************/
/* Function calculate_view                                                   */
/*****************************************************************************/
void calculate_view(short value[], pfCoord *view) {

  float sh, ch, sp, cp, sr, cr;


// calculate (pitch)

if (view->hpr[1] + (value[3] * SBRATIO) > 90.0f)
   view->hpr[1] = 90.0f;

else
  if  (view->hpr[1] + (value[3] * SBRATIO) < - 90.0f)
     view->hpr[1] =  - 90.0f;

  else
       view->hpr[1] += value[3] * SBRATIO;

// calculate (roll)

if (view->hpr[2] + (value[5] * SBRATIO) > 90.0f)
   view->hpr[2] = 90.0f;

else
  if  (view->hpr[2] + (value[5] * SBRATIO) < - 90.0f)
     view->hpr[2] =  - 90.0f;

  else

       view->hpr[2] += value[5] * SBRATIO;
// calculate (heading)

if (view->hpr[0] + value[4] * SBRATIO > 360.0f)
   view->hpr[0] = (view->hpr[0] + value[4] * SBRATIO) - 360.0f;

else
  if (view->hpr[0] + value[4] * SBRATIO < - 360.0f)
     view->hpr[0] = (view->hpr[0] + value[4] * SBRATIO) + 360.0f;

  else
     view->hpr[0] += value[4] * SBRATIO;

  pfSinCos(view->hpr[0], &sh, &ch);
  pfSinCos(view->hpr[1], &sp, &cp);
  pfSinCos(view->hpr[2], &sr, &cr);


 // Calculate Performer Y-axis position

if (view->xyz[1] + (value[2]*ch + value[0]*sh) * SBTRATIO < -3000.0f)
         view->xyz[1] = -3000.0f;

else
  if (view->xyz[1] + (value[2]*ch + value[0]*sh) * SBTRATIO > 3000.0f)
     view->xyz[1] = 3000.0f;

  else
     view->xyz[1] += (value[2]*ch + value[0]*sh) * SBTRATIO;
```

```
// Calculate Performer Z-axis position

if (view->xyz[2] + value[1] * SBTRATIO < 5.0f)
        view->xyz[2] = 5.0f;

      else
 if (view->xyz[2] + value[1] * SBTRATIO > 1000.0f)
    view->xyz[2] = 1000.0f;

 else
    view->xyz[2] += value[1] * SBTRATIO;


// Calculate Performer X-axis position

if (view->xyz[0] - (value[2]*sh - value[0]*ch) * SBTRATIO < -3000.0f)
        view->xyz[0] = -3000.0f;

else
 if (view->xyz[0] - (value[2]*sh - value[0]*ch) * SBTRATIO > 3000.0f)
    view->xyz[0] = 3000.0f;

 else
    view->xyz[0] -= (value[2]*sh - value[0]*ch) * SBTRATIO;

} // end calculate_view()



//*****************************************************************************
// FILENAME: loadGDL.h
// PURPOSE:  This program contains the functions prototype for loading a file
//
// NOTE:
// AUTHOR:  John Goetz,
// DATE:    15 January 1994
// COMMENT:
// UPDATE:
//*****************************************************************************


#ifndef __LOADGDL_H__
#define __LOADGDL_H__

#include "pf.h"

 pfGroup* LoadGDL2(char* filename);

#endif
```

APPENDIX C: LOADING NPSGDL2 FILES INTO PERFORMER

1. PURPOSE

The purpose of this loader is to enable the use of the extensive model library that has been created using Object File Format (OFF), NPS Graphics Description Language (NPSGDL) or NPS Graphics Description Language II (NPSGDL2) in Performer applications. These three formats differ slightly and some text editing of the model description is required to convert OFF and NPSGDL files into the NPSGDL2 format. For a complete description of NPSGDL see [Ref. 15]

2. GENERAL DESCRIPTION

The loader reads a NPSGDL2 file and stores all similar geometry with the same state and same predraw callback in a pfGeode..State refers to the color, material, texture or texture environment. The predraw callback is discussed on page 69. When the state changes, the current geoset along with its state information (geostate) is attached to the current pfGeode. A new geoset and geostate is then created. If the state is changed redundantly (red, white, red, white) many times, an excess number of pfGeosets will be created and will increase the traversal time for the scene. In order to alleviate this problem it is useful to group all polygons with similar characteristics (state) together in the model file

After the entire model file has been read in, all the pfGeodes which have been created are attached to a pfGroup. A pointer to the pfGroup is returned to the main application for the user to attach to the scene. The following sample code shows how to load a file:

```
#include "loadGDL.h"
#include "pf.h"

pfGroup *G_dirt;
G_dirt = LoadGDL2("models/test_floor.gdl");
```

3. PRIMITIVES

Performer only allows for a few well defined primitives. These are points, lines, linestrips, tris (triangles), quads (quadrilaterals) and tristrips[1]. GDL allows lines, Tmeshes (triangle mesh), Qstrips (similar to tristrips except components are quadrilaterals) and polygons which may have any number of sides (sides > 2).

NPSGDL Tmeshes convert easily to Performer tristrips. Qstrips and lines were ignored since these features did not seem to be used very often. Polygons on the other hand are used extensively and the limited Performer primitives must be made to handle any polygon. For this reason all polygons are converted to Performer tristrips. This is accomplished by the loader and is transparent to the user.

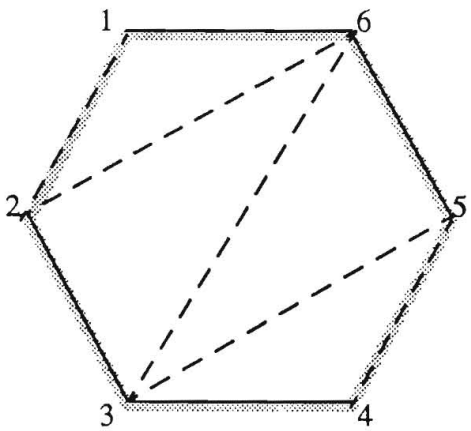


Figure C1: Converting a polygon to a trimesh

Figure C1 depicts a typical 6 sided polygon and how the loader breaks it down into a trimesh. The dashed line depicts the vertex path traversed by the loader (1, 2, N, 3, N-1, 4, N-2), and the solid lines depict the sides generated by the tristrip algorithm. The loader can convert any convex N-sided polygon

It is not apparent, but there is a difference between trimeshes and polygons converted into trimeshes. It is in the way they are indexed. Referring to Figure C1, the vertices of a

1. A tristrip is a collection of three sided polygons specified by a sequence of the three most recent vertices. Also referred to as triangle mesh

trimesh would be entered in the order (1,2,6,3,5,4). The index list (traversal order) would be NULL, meaning the vertices will be traversed in the order they were entered. The vertices for a polygon would be entered in the order (1,2,3,4,5,6) and the index list would be (1,2,6,3,5,4).

The loader can be changed to handle 3 and 4 sided polygons more efficiently, but this would require editing the model files and grouping similar polygons together to minimize changes in the primitive type. Redundant changes will create an excess number of geosets which will slow traversal time.

4. COORDINATE SYSTEM

Performer utilizes an orthogonal coordinate system defined with the Y axis to the right, Z axis up, and the X axis out of the screen. This differs from IRIS's GL (Graphics Library) coordinate system; utilized by GDL2; which has the X axis to the right, Y axis up, and the Z axis out of the screen. In order for models stored using GDL to appear correctly when rendered using Performer, the loader changes the order of the vertices from XYZ in GDL to YZX in performer.

5. MATRIX TRANSFORMATIONS

The NPSGDL2 commands for manipulating the model view stack (scale, rotate, translate, pushmatrix, popmatrix, loadmatrix) are used by the loader to transform vertices to their desired location. Polygon and/or vertex normals are also transformed, but only by the rotation matrix.

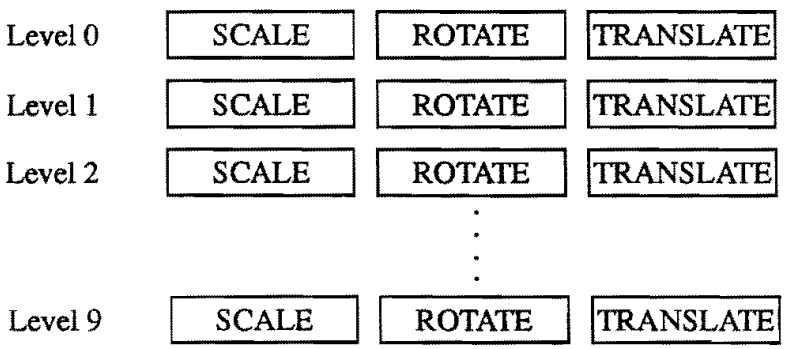


Figure C2: GDL Loader Matrix Array

The loader has an array of ten sets of matrices. Each matrix set consists of three 4x4 matrices, one for scale, rotate and translate (see figure C2). When any or all of the matrices are defined, they are multiplied together in the order S x R x T and the resultant matrix is placed at the top of the array (Level 0). All vertices that follow are transformed by the top level matrix prior to being saved in the pfGeoset. When a pushmatrix is encountered the array is incremented. All three matrices of the new level are loaded with the identity matrix and they are ready for assignment. A popmatrix decrements the array and the previous matrix is returned. The loader will not allow a user to push more than 10 matrices nor pop more matrices than were pushed.

a. LOADING A MATRIX

The rotation matrix only allows for single axis rotations. If multiple rotations are desired then the loadmatrix command can be used to enter a 4 X 4 homogeneous transformation matrix. The matrix is entered in column vector format (translations in the last row); common in graphics; not row vector format (translations in the last column); which is common in robotics.

This matrix is physically stored in the rotate matrix position of the current matrix level. In order to correct for the coordinate system differences between GDL2 and Performer, the loader transforms the matrix to correspond to the Performer coordinate system.

6. RENDERING ATTRIBUTES

Specifying the appearance of primitives may be accomplished using setcolor, setmaterial or settexture. If textures are used, then a texture environment must also be specified. Texture coordinates may be assigned in the polygon description of the model or automatically generated by specifying a texture generating algorithm.

a. TEXTURE COORDINATE GENERATION

The GL graphics commands for automatically generating texture (texgen) coordinates have been placed in a predraw callback for pfGeodes. Each time the texgen plane equation is changed, a new geode is created and attached to the tree with a new callback. For a complete description of GL's texgen function, see [Ref. 16].

7. COMMENTS

Comments are allowed in the model files in either C notation /* */ or C++ notation //.

8. FUTURE WORK

The following GDL2 commands are not recognizable by the loader:

- deflight, setlight
- deflmodel, setlmodel
- decals
- defline
- defqstrip
- loadunit

The following GDL2 commands are recognizable by the loader, but have not been implemented at this time:

- bounds
- origin

9. TESTING

The loader has been tested successfully on several different GDL2 files. However, this does not mean it has survived rigorous and thorough testing. If anyone using this loader should have difficulties, please send email to goetzjr@scs.usna.navy.mil with a description of the problem and it will be investigated.

10. SOURCE CODE AND LIBRARY

The source code for the loader is located in /n/elsie/work3/goetzjr/loadgdl/ and the library is libpfgdl.a. The library utilizes a pointer repository to reserve memory for

instantiating template objects. In order for this repository to be visible by an application, the entire path name of the repository must be specified in the makefile. An example of this is:

```
C++FLAGS = $(INCLUDES) pte.C -ptr/n/elsie/work3/goetzjr/loadgdl/ptrepository
```

LIST OF REFERENCES

1 Schue, C. A. III, *Simulation of Tripod Gaits for A Hexapod Underwater Walking Machine*, M.S. Thesis, Naval Postgraduate School, Monterey, CA, June 1993.

2 Kwak, S. H. and McGhee, R. B., "Rule-based Motion Coordination for a Hexapod Walking Machine," *Advanced Robotics*, Vol. 4, No. 3, pp. 263 - 282, 1990.

3 Byrnes, R. B., *The Rational Behavior Model: A Multi-Paradigm, Tri-level Software Architecture for the Control of Autonomous Vehicles*, Ph.D. Dissertation, Naval Postgraduate School, Monterey, CA, March 1993.

4 McGhee, R. B., "Vehicular Legged Locomotion," in *Advances in Automation and Robotics*, Volume 1, ed. by G. N. Saridis, JAI Press Inc., pp. 259 - 284, 1985.

5 Manko, D. J., "A General Model of Legged Locomotion on Natural Terrain", Kluwer Acedemic Publishers, Boston, 1992.

6 Akizono, J., "Development on Walking Robot for Underwater Inspection," *Advanced Robotics 1989*, Springer - Verlag, pp. 652 - 663, 1989.

7 Brutzman, D. P., "From Virtual World to Reality: Designing an Autonomous Underwater Robot", AAAI Fall Symposium on Applications of Artificial Intelligence Real-World Autonomous Mobile Robots, Cambridge, MA, October 23-25, pp 18-22.

8 Yoneda, K., Suzuki, K. and Kanayama, Y., "Gait Planning for Versitile Motion of a Six Legged Robot", *Procedings of IEEE International Conference on Robotics and Automation*, San Diego CA, May 1994.

9 Davidson, S. L., *An Experimental Comparison of CLOS and C++ Implementations of an Object-Oriented Graphical Simulation of Walking Robot Kinematics*, M.S. Thesis, Naval Postgraduate School, Monterey, CA, March 1993.

10 Kristiansen, K. W. VII, *A Computer Simulation of Vehicle and Actuator Dynamics for a Hexapod Walking Robot*, M.S. Thesis, Naval Postgraduate School, Monterey, CA, March 1994.

11 McMillan, S., *Computational Dynamics for Robotic Systems on Land and Under Water*, Ph.D. Dissertation, The Ohio State University, Columbus, OH, September, 1994.

12 Cooper, L., *IRIS Performer Man Pages*, Silicon Graphics, Inc., Mountain View, CA, 1992.

13 Young, R. D., *NPSNET IV: A Real-Time, 3D, Distributed Interactive Virtual World*, M.S. Thesis, Naval Postgraduate School, Monterey, CA, September 1993.

14 Brutzman, D., *An Underwater Virtual World for an Autonomous Underwater Vehicle*, Ph.D. Dissertation, Naval Postgraduate School, Monterey, CA, June 1994.

15 Wilson, K. P., NPSGDL: *An Object Oriented Graphics Description Language for Virtual World Application Support*, M.S. Thesis, Naval Postgraduate School, Monterey, CA, September 1992.

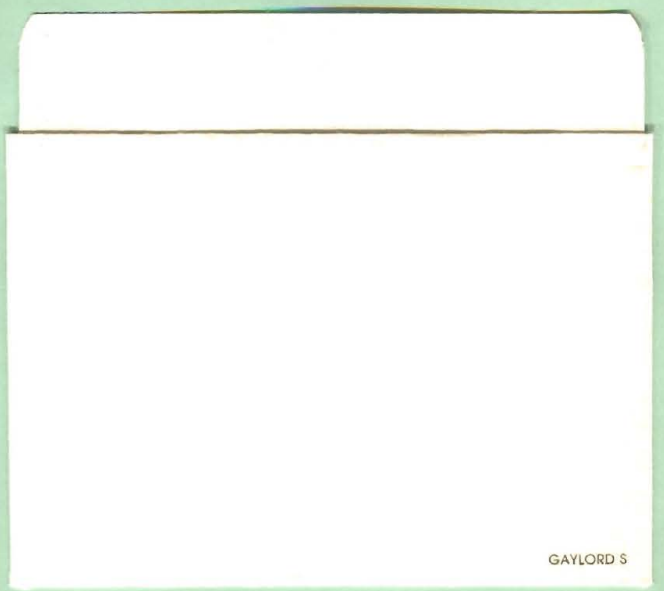
16 McLendon, P., *Graphics Library Programming Guide*, Silicon Graphics, Inc., Mountain View, CA, 1991.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Dudley Knox Library Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Dr. Ted Lewis Code 37, Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	1
4.	Dr. Robert B. McGhee Code CS/Mz Professor, Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	3
5.	Dr. Yutaka Kanayama, Code CS/Ka Computer Science Department Naval Postgraduate School Monterey, CA 93943	1
6.	Mr. Hidetoshi Takahashi Port and Harbour Research Institute Ministry of Transport 1-1, 3-Chrome, Nagase Yokosuka, Japan	1
7.	Dr. Michael J. Zyda Code CS/Zk Professor, Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	10
8.	John R. Goetz U. S. Naval Academy Math and Science Division CS Department Annapolis, MD 21402-5006	2

9.	Lt. Karl J. R. W. Kristiansen 264 Worden St. Portsmouth, RI 02871	1

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5101



DUDLEY KNOX LIBRARY
3 2768 00098865 3