(2)

AD-A196 964

NPS52-88-012

NAVAL POSTGRADUATE SCHOOL
Monterey, California

DTIC FILE COPY

THESIS

A PROTOTYPE SIMULATION SYSTEM FOR COMBAT VEHICLE COORDINATION AND MOTION VISUALIZATION

Corinne M. McConkle
and
Andrew H. Nelson

June 1988

Thesis Advisor: R. B. McGhee

Approved for public release; distribution is unlimited.

Prepared for:
USACDEC
Ft. Ord, California 93941

DTIC
ELECTE
AUG 1 6 1988
D
H

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R. C. Austin
Superintendent

Kneale T. Marshall
Acting Provost

This thesis is prepared in conjunction with research sponsored in part by contract from the United States Army Combat Developments Experimentation Center (USACDEC) under MIPR ATEC 88-86.

Reproduction of all or part of this report is authorized.

The issuance of this thesis as a technical report is concurred by:

[illegible]

ROBERT B. MCGHEE
Professor
of Computer Science

Reviewed by:

[illegible]

VINCENT Y. LUM
Chairman
Department of Computer Science

Released by:

[illegible]

JAMES M. FREMGEN
Acting Dean of Information and Policy Science

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION AVAILABILITY OF REPORT Approved for public release; Distribution is unlimited
2b DECLASSIFICATION DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S) NPS52-88-012	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School — 6b OFFICE SYMBOL (If applicable) Code 52	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000	7b ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000
8a NAME OF FUNDING SPONSORING ORGANIZATION USACDEC — 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER ATEC 88-86
8c ADDRESS (City, State, and ZIP Code) Ft. Ord, California 93941	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO | PROJECT NO | TASK NO | WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)
A PROTOTYPE SIMULATION SYSTEM FOR COMBAT VEHICLE COORDINATION AND MOTION VISUALIZATION

12 PERSONAL AUTHOR(S)
McConkle, M. and Nelson, Andrew H.

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Master's Thesis	FROM ______ TO ______	1988 June	191

16 SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Autonomous vehicle; Artificial intelligence; Computer simulations; Rule based systems; Robotics

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

This thesis develops a prototype rule based command and control system for units of autonomous tactical vehicles. By applying artificial intelligence techniques, tactical coordination of multiple autonomous land vehicles is also accomplished. This study identifies the requirements for such a system and provides a prototype system with a sophisticated computer graphics simulator as a testing facility for future follow on research.

This study was a joint research project. Andrew H. Nelson was responsible for the rule system modeling on the LISP machines and Corinne McConkle was responsible for the real-time graphics motion visualization on the IRIS workstation. The networking software was developed jointly. [illegible]

20 DISTRIBUTION AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL Prof. Robert B. McGhee	22b TELEPHONE (Include Area Code) (408) 646 - 2095 — 22c OFFICE SYMBOL 52Mz

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

☆ U.S. Government Printing Office: 1986—606-24.

Approved for public release; distribution is unlimited.

A PROTOTYPE SIMULATION SYSTEM FOR COMBAT VEHICLE COORDINATION AND MOTION VISUALIZATION

by

Corinne McConkle
Lieutenant, United States Navy
B.A., Whittier College, 1976
and
Andrew H. Nelson
Captain, United States Marine Corps
B.A., University of New Mexico, 1982

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

June 1988

Authors: [illegible]
Corinne McConkle, Andrew H. Nelson

Approved by: [illegible]
Robert B. McGhee, Thesis Advisor

[illegible]
Michael J. Zyda, Second Reader

[illegible]
Vincent Y. Lum, Chairman
Department of Computer Science

[illegible]
James M. Fremgen,
Acting Dean of Information and Policy Sciences

ABSTRACT

This thesis develops a prototype rule based command and control system for units of autonomous tactical vehicles. By applying artificial intelligence techniques, tactical coordination of multiple autonomous land vehicles is also accomplished. This study identifies the requirements for such a system and provides a prototype system with a sophisticated computer graphics simulator as a testing facility for future follow on research.

This study was a joint research project. Andrew Nelson was responsible for the rule system modeling on the LISP machines and Corinne McConkle was responsible for the real-time graphics motion visualization on the IRIS workstation.

QUALITY INSPECTED 2

Accession For

NTIS GRA&I	✓
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	

Dist	Avail and/or Special
A-1	

"Original contains color plates: All DTIC reproductions will be in black and white"

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. BACKGROUND

The development of a prototype rule based command and control system for units of autonomous tactical vehicles is the major objective of this work. This study is motivated by two important factors. First, an automatic pilot has been developed that can drive an autonomous land vehicle in real time using a dynamic path planning approach [Ref. 1]. Second, it can be quickly recognized that a successful military application of this technology must include an enhanced mission capability that addresses the coordination and control of multiple autonomous vehicles operating as a combined unit. This study investigates the feasibility of applying artificial intelligence techniques to develop an automated command and control system to allow tactical coordination of multiple autonomous land vehicles. The purpose of this study is to identify requirements for such a system and to provide a prototype system with a suitable testbed facility for future follow on research.

Techniques developed in this study pertaining to this area of research have immediate application to the areas of research currently being studied in AI and to the ongoing existing development work at FMC [Ref. 1] for military application of autonomous vehicles. Further, the prototype system developed in this study can provide a cost effective testing environment for new algorithms as they are developed.

B. ORGANIZATION

Chapter II provides a summary of some of the previous research work relating to this study. It includes previous vehicle simulation and display systems, off-road vehicle simulations using the DMA digital terrain elevation database, models for autonomous vehicle drivers, and doctrine for vehicle cooperation. It also provides a brief discussion on the hardware and software computational tools used in the simulation.

Chapter III provides a detailed description of the three major research areas of this study, terrain modeling, vehicle characteristics and dynamics, and the division between the graphical and logical aspects of the problem. The vehicle characteristics and dynamics section describes the general characteristics of the military vehicles used in this simulator. It also provides the mathematical models used for vehicle acceleration, steering, and *bounce*. The terrain model describes the simulated environmental characteristics in which the military vehicles operate. That chapter also presents the command and control subsystems and rules for collision avoidance, speed determination, direction determination, and stationing.

Chapter IV presents the structure of the prototype simulation system. The terrain and vehicle simulation, the driver and commander simulation, the hardware, software, and communication interfaces for the graphics simulator and the autonomous vehicle controllers are discussed in this chapter. A users guide is included to facilitate operation of the simulator.

Chapter V is a review and analysis of the performance of the autonomous vehicle simulator.

Chapter VI is a summary of the research contributions of this work and possible extensions for future research.

This study was a joint research project. Andrew Nelson was responsible for the rule system modeling on the LISP machines and Corinne McConkle was responsible for the real-time graphics motion visualization on the IRIS workstation.

II. SURVEY OF PREVIOUS WORK

A. INTRODUCTION

The majority of the autonomous vehicle research projects currently in existence share two common characteristics. The primary shared characteristic is that they are computer based. Secondly, because of the high cost of implementing physical platforms, they rely heavily on the use of computer aided graphics and animation.

This chapter provides a basic background in autonomous vehicle research and surveys previously implemented computer-aided test platforms. A discussion of tactical considerations implemented in the scope of this thesis is provided as well as a short synopsis of the hardware and software tools used in this study.

B. PREVIOUS VEHICLE SIMULATION AND DISPLAY SYSTEMS

Previous vehicle on-road simulation and display systems developed at the Naval Postgraduate School include: an autonomous steering system by Tan [Ref. 2] and a speed control system by Dolezal [Ref. 3].

1. Autonomous Steering System

In [Ref. 2], Tan investigated a new technique for autonomous vehicle steering control through an *out-the-windshield* graphics simulation model. The hypothesis investigated assumes that human drivers subconsciously divide their route into small, interconnected line-of-sight segments. These segments are built dynamically while driving along the road. Drivers subconsciously set subgoals whose locations are based on factors such as vehicle speed, road surface conditions, driving experience, and environmental conditions.

In [Ref. 2], the simulation test track consists of four 400 meter straight road segments that are connected by three curved segments with an 80 meter radius. Most of the experiments were performed on a short segment of the circuit. This segment included a 200m segment of straight road followed by a curved segment and then another straight segment. Both manual driving and autopilot driving were tested. The simulation stops when either vehicle crashes or successful navigations of turns.

Manual steering and autopilot (proportional navigation, and pursuit navigation) steering techniques were modeled and tested. The manual steering method allows the operator to manually steer the vehicle by observing the out the window view and using lateral movements of the mouse. In the autopilot steering mode, the vehicle steers toward target points in the center of its driving lane. As the vehicle proceeds down the road, new target points, or subgoals, are selected. The mathematical steering models used are described in [Ref. 2].

For manual steering, the operator was easily able to navigate a turn with vehicle velocity of 50km/hr. When the velocity was increased to 100km/hr, the operator's ability to navigate the curves deteriorated significantly. At 150km/hr, manual steering was not possible. With pursuit navigation, the vehicle was able to remain in the center of the road while navigating a corner at 50km/hr and 100km/hr. At 150km/hr, the vehicle was not able to remain on the road. Proportional navigation is the same model as pursuit navigation with an additional gain term included. With this model, the vehicle was able to navigate the curve at 150km/hr.

2. Speed Control System

examines the longitudinal speed control and braking of a vehicle while mimicking human control. Research into the behavioral aspects of human driving may provide insights into future autonomous vehicle control research.

The simulation developed by Tan [Ref. 2] was modified to provide vehicle control through manual methods or by an autosteer/cruise control system. This was facilitated through a network of two IRIS workstations. One workstation shows the navigator's display and the other represents the driver's display.

Several different control modes for vehicle operation are incorporated in the simulation. The user can control the vehicle from either IRIS workstation manually, by autosteer or cruise control, or any combination of the three. The operator can change modes at any time from either display. The driver's display includes a dashboard with operating instructions, instrumentation, and an out the window view of the road and scenery. The road and scenery consists of a track with intersections, stop signs, speed limit signs and a semaphore.

During program execution, vehicle coordination, vehicle distance with respect to the start of the circuit, vehicle velocity, and vehicle braking information are continually transmitted from the driver's display to the navigator's display. The navigator's display transmits commanded velocity, vehicle braking control, vehicle steering, and operating mode to the driver's display.

As the vehicle approaches the semaphore or stop sign, the driver or autopilot adjusts the vehicles speed and uses braking as necessary to stop at the desired location. Braking deceleration is a function of brake pedal depression and the braking provided by the engine. It was found that, when using manual steering, the human driver applied hard braking late and had a difficult time stopping at the desired location. This was attributed to a lack of references normally used by human drivers to perceive motion and judge distances. In all tests with the autopilot, the autopilot applied hard breaking early and always stopped prior to entering the intersection.

C. TERRAIN SIMULATIONS

Although numerous examples of graphical terrain simulations abound in the literature [Ref. 4], for military considerations this study focuses upon specific applications of DoD information to terrain simulators. Previous simulations at the Naval Postgraduate School incorporating the Defense Mapping Agency Digital Terrain Elevation Data include: a real-time simulation of the Fiber Optically Guided Missile (FOG-M) [Ref. 5], and an Interactive, Networked, Moving Platform Simulator [Ref. 6]. Both simulators were implemented on a Silicon Graphics, Inc. IRIS graphics workstation.

1. FOG-M Simulator

The FOG-M simulator is a prototype flight simulation study designed to model the performance of the Army's Fiber Optically Guided Missile. The simulator displays a dynamic, three dimensional, out-the-window view of the terrain the missile is flying over. Interactive control of the missile camera angle, direction, speed, and elevation is accomplished through the use of a mouse and dial box.

The source of data used to represent the terrain is the Digital Terrain Elevation Database from the Defense Mapping Agency. The data represents a 35 kilometer by 36 kilometer area of terrain at Fort Hunter Liggett, California, and vicinity. The 16 megabyte digital terrain database is stored on a Digital Equipment Corporation (DEC) VAX 11/785. A subset of the master terrain database is created as a binary file and transferred to the IRIS disk storage area based on the designated missile flight area.

The input terrain source file is created off-line. The operator cannot change the flight area during program execution. The information needed to display the terrain is stored in two global arrays. The first is a five-dimensional array that stores the values of the coordinates for each triangular polygon that makes up the terrain. The second array stores the color map indices for each of the terrain's grid squares. The two triangles that make up each grid square are drawn in a different shade of green to give the terrain a checkerboard effect. The checkerboarding enhances the simulation of motion over the terrain.

2. Interactive, Networked, Moving Platform Simulators

The moving vehicle simulator (VEH) is a follow-on study to the FOG-M simulator. The moving vehicle simulator can be operated in one of two possible modes: stand alone mode or networking mode.

The stand alone mode models vehicle motion over terrain with limited vehicle dynamics. A three-dimensional, out-the-window view (as seen from the driver's position) of the terrain and other vehicles is displayed. The operator can select which vehicle's viewing volume to display. The vehicle selected is designated the *driven vehicle*. The operator can change the view shown on the operator's screen by changing driven vehicles. In the networking mode, the moving vehicle simulator provides realistic targets for the FOG-M simulator.

The VEH Simulation uses the elevation data in a digital terrain database provided by the Defense Mapping Agency (DMA) to draw the three-dimensional scene. The terrain model is similar to the model used in the FOG-M simulator with a few noteworthy exceptions. The terrain elevation data file is reformatted to match the two-dimensional array used to store it during program execution. Data points for ten lengths of ten kilometers are stored a row at a time, from west to east along a row's length, and from south to north, going from row to row. This matches the C compiler storage mapping function for two-dimensional arrays.

The ten kilometer by ten kilometer area of missile flight is sectioned into hundred meter grid squares, with each square consisting of two triangles. The triangles are used to construct a colored, three-dimensional terrain display. An external program is used to convert the elevation height values from feet to meters and scale the terrain data from the master data file.

D. MODELS FOR AUTONOMOUS VEHICLE DRIVERS

1. The FMC Autonomous Land Vehicle

A system architecture for an autonomous land vehicle has been developed at FMC Corporation, Central Engineering Laboratories [Refs. 1,7]. Currently, the FMC architecture consists of Planner, Observer, Mapmaker, Pilot, and Vehicle Control subsystems. A digitized map of terrain elevation and features is used by the Planner to create a path from start to goal point. That path is communicated to the Observer. The Observer transforms the plan into polygonal segmented path regions and goal directions. The Observer then communicates the previous, current, and next segment of the plan to the Mapmaker. An obstacle detection sensor communicates obstacle data to the Mapmaker, which then creates the *pilot map*. The *Reflexive* Pilot then processes the pilot map. The purpose of the Real-Time Reflexive Automatic Pilot is to generate vehicle control commands that direct the vehicle from the starting point to the final destination without hitting unforeseen obstacles or leaving the vicinity of the planned path. The pilot, whose input includes a map, a general goal direction, maximum-allowed vehicle forward velocity, and current vehicle forward and rotational velocities, outputs a target turn radius and translational speed command to the vehicle controller. The pilot discussed here has some similarities with other pilot systems reported in the literature [Refs. 8-10]. The FMC pilot, however, is able to generate and select subgoals in real time. It is also one of the few to take into account vehicle dynamics into its local path planning. Field tests have demonstrated that the pilot is able to guide a 10-ton M113A2 armored personnel carrier on a 1 mile course around randomly placed obstacles without crossing the outer borders of the mission plan at 5 mph.

2. The Martin Marietta Autonomous Land Vehicle

The purpose of the Martin Marietta Autonomous Land Vehicle is to provide a mobile platform to integrate and demonstrate strategic computing technologies. The primary emphasis of the program is on perception and reasoning with minimal research being conducted in the areas of control and physical vehicles [Ref. 11].

The ALV conceptually consists of the physical vehicle, sensors, and control subsystems. The Martin Marietta ALV is an eight wheeled all-terrain vehicle with a fiberglass shell. The fiberglass shell houses the internals of the environmentally controlled testbed. The ALV sensor system (Perception Subsystem) makes use of an RCA color video CCD matrix TV camera, and a laser range scanner. The Reasoning Subsystem requests scene models from the Perceptions Subsystem and converts them into smooth trajectories that can be passed to the pilot to drive the vehicle. The pilot then converts the intervals of a trajectory into steering commands for the vehicle.

The ALV was demonstrated on November 5 1987. The ALV performed an autonomous navigation run over a course length of 4.2 kilometers, achieving a top speed of 20 kilometers per hour, and an average course speed of 14 km/hr. It successfully avoided all obstacles on the road on both the start and return legs of the course [Ref. 12].

E. DOCTRINE FOR VEHICLE COOPERATION

1. Organization

Military vehicles, by the nature of their requirements, are employed as units, and operate in conjunction with other units. The smallest tactical unit of vehicles common to ground forces is a *platoon* usually consisting of from four to six vehicles. Larger tactical units are constructed in a hierarchical manner from this basic unit of organization; i.e., *companies* are composed of platoons, *battalions* are composed of companies, *regiments* are composed of battalions, etc. To further compound the complexities of their employment, military vehicles are operated in extremely complex environments. Military vehicles must frequently traverse difficult terrain under adverse conditions. These adverse conditions include: inclement weather, movement at night, and hostile conditions. Because of these complexities, a large body of doctrine and knowledge has evolved to allow military personnel to develop the skills necessary to effectively employ military vehicles to accomplish specific tactical goals and objectives. These skills can be collectively categorized under the military subject *command and control.*

Individual interpretations of the functions that constitute command and control vary among military commanders, but accepted basic doctrines have been established although the degree of their considerations varies [Ref. 13]. A discussion of basic tactical considerations is provided below.

2. Command and Control

When an operational order has been received by a commander, the use of available time is planned. The commander uses a planning sequence called "reverse planning," starting with the last action for which a time is specified and working back to the receipt of the order. During this stage, an analysis of the terrain and the friendly and enemy situation is conducted. From this, a preliminary plan of action for accomplishing the mission is formulated. This plan is tentative and frequently changes.

Once a preliminary plan is developed (largely involving planning of available time), a route is selected and then a schedule for reconnaissance and coordination with adjacent and supporting units is developed. The reconnaissance is conducted, at which time the estimate of the situation is completed. Final coordination with adjacent and supporting units is then accomplished. Effects of the terrain on the preliminary plan are incorporated as necessary. A control point, called the *vantage point,* is selected to orient subordinate unit leaders. The vantage point is a clearly recognizable feature of the local terrain.

After completing the reconnaissance, the final plan of action is formulated based upon a problem-solving process refered to as *the estimate of the situation.* This is a method of selecting the course of action which offers the greatest possibility of success and is the goal of the estimate of the situation process. Available courses of action are considered by unifying the factors of the mission, enemy situation, terrain and weather, and tactical assets available and rejecting those courses of action that would fail to satisfy the stated goal. A general definition of these unifying factors are briefly stated below.

3. Mission

The mission is a clear, concise, and simple statement of the task to be performed. It is the basis for all actions of the unit until it is accomplished.

4. Enemy Situation

The most important information about the enemy situation is strength, location, composition, type of weapons, disposition, tactical methods, and recent actions.

5. Terrain and Weather

The terrain and weather affect all plans and actions. The weather, both present and predicted, have an effect upon visibility, movement, and fire support. The military aspects of terrain are:

a. Key Terrain

A key terrain feature is any locality or area the seizure or control of which gives a marked advantage to either opposing force. This advantage lies generally in terrain which affords good observation and fields of fire.

b. Observation and Fields of Fire

Observation is the ability of the unit to see the enemy locations. It assists in gaining information of the enemy, and in accurately directing fire on the enemy. Fields of fire are the areas that a weapon or group of weapons can cover and are essential to the effective employment of direct fire weapons. This factor is considered for both opposing forces.

c. Cover and Concealment

Cover is the protection from enemy fire. Concealment is the hiding or disguising of a unit and its activities from observation.

d. Obstacles

Obstacles are natural or artificial terrain features which stop, delay, or restrict military movement. Obstacles can either help, or hinder a unit, depending upon location and nature. In general, obstacles perpendicular to the direction of movement favor defending forces, while those parallel to direction of movement can favor attackers by protecting flanks.

e. Avenues of Approach

An avenue of approach is a terrain area that permits a route of movement for a unit, providing ease of movement, cover and concealment, favorable observation and fields of fire, and adequate maneuver room.

6. <u>Tactical Assets Available</u>

Unit capabilities are considered as well as available support (in terms of supressive or screening fires, etc.).

7. <u>Completing the Plan</u>

Once the plan is formulated, subordinate units are communicated their operational orders. These units perform the same cycle of planning. The mission is executed, and the results reported.

8. Automation of the Cycle of Planning

Although the above description is general in its discussion of the command and control process, typical field manuals suggests the course of action that is pursued in this study for automation. A "reverse planning" approach lends itself very well to backward-chaining techniques already developed and applied in artificial intelligence. Factors for consideration can be translated to rules of action; facts and rules can be analyzed to arrive at the specific task sequences necessary for goal accomplishment where the specific method of subgoal accomplishment is not known beforehand (forward-chaining). Unification of those factors to decide possible courses of action tend to suggest backward-chaining while choosing the best possible course tends to suggest a branch and bound search using constraints such as speed, minimum predicted losses, or minimum expenditures of material.

F. COMPUTATIONAL TOOLS

1. Hardware

A wide variety of hardware is available to address specific problem areas in this study. A general discussion of their properties and areas of application follows.

a. Silicon Graphics, Inc. IRIS

The graphics motion visualization portion of the autonomous motion simulator is implemented on a Silicon Graphics, Inc. IRIS-2400 Turbo high performance color graphics workstation [Ref. 14]. The workstation uses custom VLSI chips to provide hardware clipping and matrix transformations. This high speed, pipelined architecture allows the performance of viewing, modeling, projection and display device transformations at a much greater rate than would be possible in software.

The graphics hardware can be conceptually depicted as three pipelined components: the applications/graphics processor, the geometry pipeline, and the raster subsystem. The geometry pipeline and the raster subsystem are controlled by the applications/graphics processor.

The IRIS provides a double buffer display system with a resolution of 1024 by 768. This is essential to produce the smooth animation necessary for motion simulation studies.

b. LISP Machines

LISP machines are microprogrammed computers, with a very large microcode memory (typically 16k by 64 bits) [Ref. 15]. Lisp source code is compiled to a virtual machine code (fasl), and the microcode interprets the fasl. To allow interpreted execution of LISP source code, there is a compiled interpreter program that is always resident in the system. Stack, virtual memory management, and garbage collection are all implemented in microcode. A microcode compiler is resident on the system, allowing easy modification or extension of the LISP environment.

The data paths of a typical LISP machine are shown in Figure 2-1. A register file drives one input of the ALU, and the other ALU input is driven by a bus. Items on the bus include a second register file, a stack cache, Virtual Memory Address register, Memory Data register, the LISP macrocode program counter, and the macrocode instruction buffer. The ALU output drives the machine's main data bus. In parallel with the ALU is a shifter/masker that also drives the main bus. Each machine instruction uses either the ALU or the shifter/masker to perform its operation. LISP machines use tagged data, usually implemented as the top few bits of the data word, to provide hardware support for data type-checking. Typical LISP machines support 16 - 34 data types. A separate tag register is present on some LISP machines.

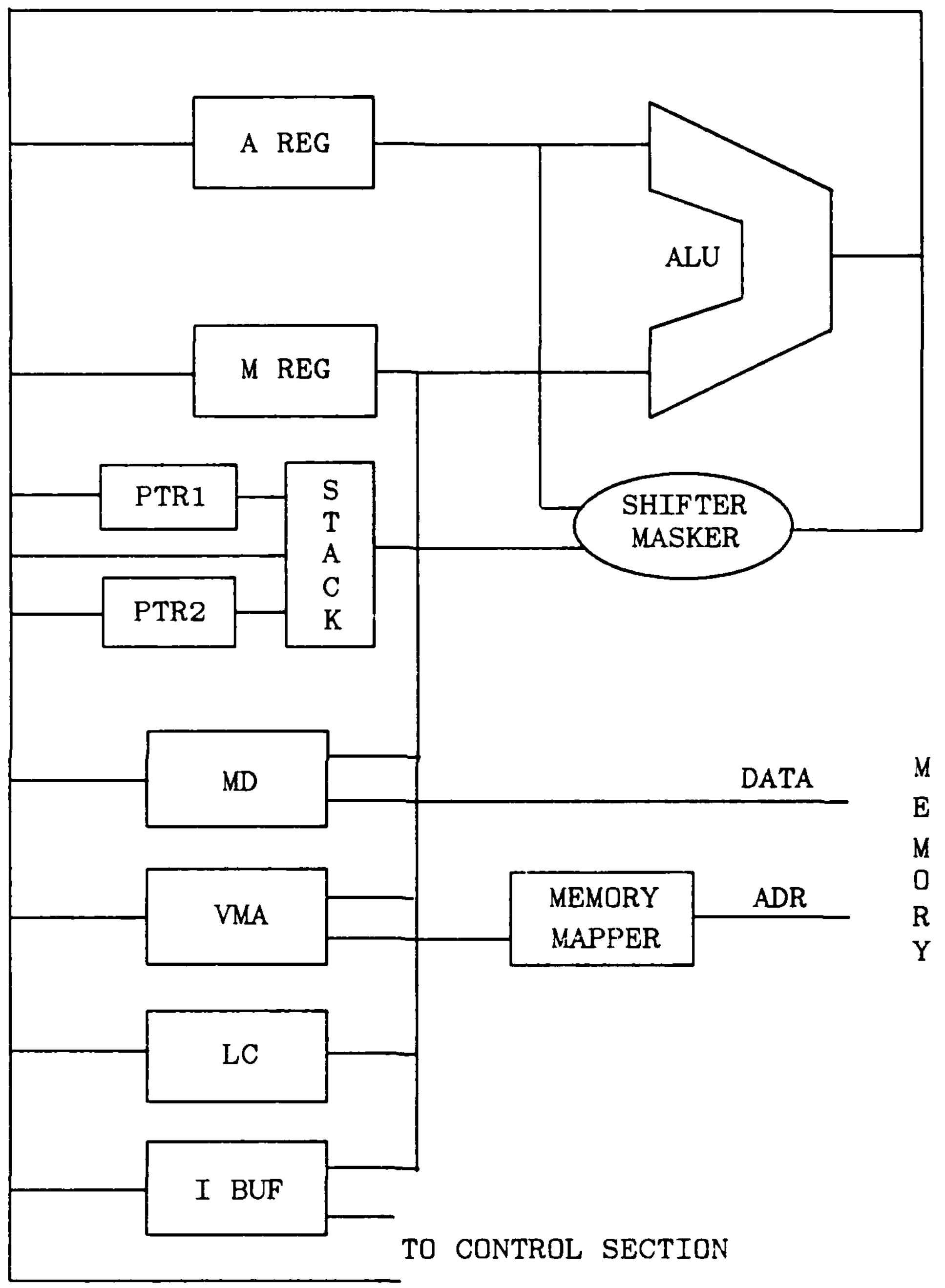


Figure 2.1 Lisp Machine Data Paths

c. TI/Explorer LISP Machine

The Texas Instruments Explorer LISP machine consists of a LISP processor, 2-16 Mbytes of DRAM memory, a 5.25-inch 112 Mbyte Winchester disk, a Local Area Network (LAN) interface, a high-resolution bit-mapped display, a mouse pointing device, and a keyboard. The 32-bit NuBus [Ref. 16] is the main system bus. A separate 32-bit local bus connects the processor, memory, and display controller. The processor uses a 32-bit tagged data path (25 data and 7 tag bits). The 25-bit pointers provide a 128 Mbyte virtual address space. Memory access can bypass the virtual address mapping hardware, allowing full 32-bit (4 Gbyte) logical address space.

d. Symbolics 3600 Lisp Machine

The SYMBOLICS 3600 LISP machine consists of a dedicated 36-bit LISP processor, 28 bit word addressed demand-paged virtual memory, a high-resolution bit-mapped display, dedicated console microprocessor for handling keyboard and mouse input, a MC68000-based front-end processor (FEP), and secondary storage utilizing standard Winchester technology [Ref. 17]. In addition, the system possesses a 10-Mbit/sec Ethernet transceiver and interface.

e. Ethernet

Each machine discussed above is designed to be linked in an Ethernet based LAN [Ref. 18]. Ethernet (IEEE Standard 802.3) is a hardware standard that implements Layers 1 and 2 of the Open Systems Interconnection standard. Ethernet cards installed on the above machines handle the physical and link layer tasks between the transmission medium, which is coaxial cable.

2. Software

A wide variety of software tools are available to address specific problem areas in this study. A general discussion of their properties and areas of application follows.

a. C

The C programming language is a mid-level language that functions both as a systems language and an application language [Ref. 19]. The dominant characteristic of C is its uncommon power. This power comes from two specific properities, function-orientation, and weak typing. C is function-oriented in that programs consist solely of a series of functions, which in turn, are composed of functions. In C, functions also serve as procedures. C is weakly typed, all values and data variables can be coerced into a wide variety of data representations and even into addresses. These characteristic allows programmers to start programming at the system level and iteratively compose larger functions from previously designed ones to create procedural abstractions that implement complex problem solving algorithms.

b. Lisp

Lisp developed in the late 1950s out of the needs of artificial intelligence programming. Lisp is usually characterized by four properties, its ability to manipulate symbols and lists, the ability to apply functions, extensibility, and its interactive interpreter [Ref. 20]. Lisp manipulates lists of atomic symbols. In fact, Lisp programs are lists. Because of this, Lisp has the ability to literally create and manipulate programs written in Lisp. Lisp programs recursively and iteratively execute by applying functions to a list of arguments. Lisp is extensible, which means that the language itself can be extended using the basic Lisp primitives. Lisp systems are usually interactive interpreters. Programmers interact with the Lisp interpreter by typing in function applications. The Lisp system interprets them and prints out the result. Lisp has been standardized to Common Lisp [Ref. 21].

c. Flavors

The flavor system is a Lisp extension for defining and creating active objects, that is, objects that can receive messages and act on them [Ref. 7]. A *flavor* is a class of active objects. One such object is called an instance of that flavor.

There are two primary characteristics of a flavor: the set of messages an instance can receive and the set of state variables of an instance. For each message an instance can receive, it has a corresponding method or function to invoke. That method is shared by all instances of the flavor. Every object of a given flavor has the same set of instance variables, but the values of those instance variables vary from object to object. Methods are the only functions that can manipulate those objects [Ref. 22]. Flavors can inherit methods and instance variables from other flavors.

d. TCP/IP

The Transmission Control Protocol (TCP) and Internet Protocol (IP) is an interface requirement for the Defense Data Network [Ref. 18]. It was developed at Stanford University and is used in Arpanet. IP is a simple internetwork communication protocol that sends and receives packets. It is considered to be Layer 3 of the Open Systems Interconnection standard. It sends data among many types of networks including Ethernet LAN. TCP is a Layer 4 tansport protocol permitting two hosts to establish a connection, exchange data, and terminate the connection when finished. The communications packages developed in this study implement Layer 7 of the Open Standard Interface (the application layer) utilizing the presentation and session protocols (Layers 5 and 6) of the respective host machines.

e. KEE

KEE is an expert system shell that resides on both the TI Explorer and Symbolics LISP machines. KEE uses a frame-based representation of objects and their attributes to form a knowledge base. The knowledge base is managed by an extensive truth maintenance system that incorporates forward and backward chaining of rules and methods to accomplish inferencing. KEE's extensive programming tools allow rapid prototyping of artificial intelligence applications [Ref. 23].

f. Prolog

Prolog stands for "programming in logic" and prolog programs are expressed in the form of propositions that assert the existence or non-existence of a desired result [Ref. 24]. Prolog programs implement Horn clause forms that describe objects, properties of objects and the relationships between those objects and their properties. A Prolog programs consists of facts and rules about objects and their relationships. Prolog makes deductions based upon first order predicate logic. Inferences are made through the process of unification of variables in facts and rules, which is invoked by a query [Ref. 25].

g. Chaosnet

Chaosnet is a local network for communication among a group of computers over short distances. The name Chaosnet refers to the lack of any centralized control element in the network. Chaosnet was originally developed in 1975 by the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology as the internal communications medium of early LISP machine systems [Ref. 26].

Chaosnet consists of hardware and software, which, while logically separate, are designed for each other. The hardware provides a carrier-sense multiple-access structure compatible with Ethernet interfacing hardware. Network nodes contend for access to the ether over which they transmit addressed packets to other network nodes. The software defines the higher-level protocols in terms of packets.

Chaosnet support for both types of LISP machines consists of a set of Lisp functions and data-structure definitions in the *chaos flavor*. The type of data structure support incorporated for this study is the *connection.* There are two process which belong to the Chaosnet Network Control Program. The *receiver* process looks at packets as they arrive. Control packets are processed immediately. Data packets are put on the input packet queue of the connection to which they are directed. The *background* process wakes up periodically to do retransmission, probing, and processing *connection interrupts.*

G. Summary

This chapter provides the background of information necessary for the development of the scope of the project defined in Chapter Three. It includes discussions of previous work done in graphic simulations and autonomous vehicle systems. Tactical considerations necessary for military applications in artificial intelligence are discussed. A brief overview of hardware and software tools necessary for implementation of the research is also provided.

III. DETAILED PROBLEM STATEMENT

A. INTRODUCTION

The development of expert system based coordination algorithms for tactical units of autonomous vehicles is the major objective of this study. The second objective is to develop the software necessary to create motion simulation of the system using realistic vehicle dynamics over a computer generated terrain model. For purposes of this study, the prototype system developed closely follows the model of the FMC autopilot [Ref. 1].

This chapter treats three major areas of research: the terrain model, vehicle characteristics, and the division between the graphics and logical aspects of the study.

B. TERRAIN MODEL

The terrain model developed for this study possesses the following characteristics: adequate representation of terrain, the ability to reference positions using military grid coordinates, and the ability to provide a virtual tactical environment using the available computer and graphics display assets of the Naval Postgraduate School.

1. The Terrain

The terrain information is modeled from a Defense Mapping Agency (DMA) digital terrain elevation database (DTED) for Fort Hunter-Liggett, California. It is based on DMA forty foot interval contour map products. The database is a special product with a resolution eight times greater then Level 1 DTED. There are 6400 data points per square kilometer in the database. The area covered by the database is bounded by latitudes 36 05' 00'' (northern boundary) and 35 50' 00'' (southern boundary) and longitudes 121 04' 30'' (eastern boundary) and 121 20' 30'' (western boundary). This represents an area thirty-six kilometers wide by thirty-five kilometers high. This area is represented in terms of Universal Transverse Mercator (UTM) coordinates by an east (X coordinate) of 10SFQ41000 to 10SFQ77000 and a north (Y coordinate) of 10SFQ60000 to 10SFQ950005 .

2. The Terrain Simulation

The terrain model used in this study was developed by, and described in detail in [Ref. 5]. It vas further refined and enhanced in a follow-on study by [Ref. 6].

The data in the digital terrain elevation database is stored as an array of data points representing the terrain of Fort Hunter Liggett, California. Each data point, representing one elevation datum, is a sixteen bit integer. The lower thirteen bits represent one of 8192 possible terrain elevation values. The upper three bits represent one of seven possible vegetation height values: less than one meter, one to four meters, eight to twelve meters, twelve to twenty meters, greater than twenty meters, and no data available.

Although the data points are sampled at 12.5 meter intervals, early tests [Ref. 5] showed that the use of this resolution resulted in a very slow graphical display because of the amount of detail involved. To ensure an adequate frame update rate, a one hundred meter resolution was selected for implementation. This provides adequate resolution without sacrificing realism due to a low frame update rate.

A forty foot contour map is the basis for the three-dimensional terrain images generated by the simulation study. The three-dimensional contour is constructed from colored triangular polygons. The simulator uses a ten kilometer by ten kilometer portion of the terrain database that is divided into hundred meter grid squares. Each of the grid squares is made up of two colored triangles. The color of the triangles is determined by the vegetation codes. Brown triangles represent terrain with vegetation less than one meter high. Since most of the terrain in the Fort Hunter Liggett area consists of grass covered valleys and high ground that is all below the tree line, the result is a map with brown valleys and green ridgelines. There are sixteen color intensities used to shade the map. The color intensity levels are determined by the terrain elevation. High intensity colors represent higher elevations and low intensity colors represent lower elevations.

Lambert's Cosine Law model for shading, combined with a checkerboard artificial texture, provide a realistic display within the computational time restraints of the IRIS. The checkerboard effect is implemented by averaging the shades for the two triangles which make up a gridsquare. The average is used to remove the visible boundary between the two triangles and results in a single shaded grid square. Adjacent grid squares use offset color ramps for shade computations. This allows the shade of adjacent grid with identical surfaces to vary. The world coordinates of the triangle vertices are stored in a five-dimensional *gridcoordinate* array with the following indices: Z coordinate, X coordinate, upper or lower triangle, which vertex (vertexes are numbered in the order required to use backface polygon removal), and which coordinate (X,Y,X). This array is illustrated by Figure 3.1 as reproduced from [Ref. 6].

To display a frame of the display, the program selects the triangle coordinates to be drawn by first looping through the X and Z indices of the gridcoordinate array and then calling the IRIS graphics library polygon fill routine with the appropriate color. *Off-line processing of the terrain database includes: exponential scaling of raw elevation* values, converting the scaled data to metric values, and storing the modified information in the elevation data file. The exponential scaling is accomplished by the relation

$$Elev_{new} = Elev_{old}^{\sigma} \tag{3.1}$$

The scale factor used, σ, is 1.05. This scaling is done to provide an artificial terrain which is slightly more rugged than the actual physical terrain in order to make the terrain relief more obvious.

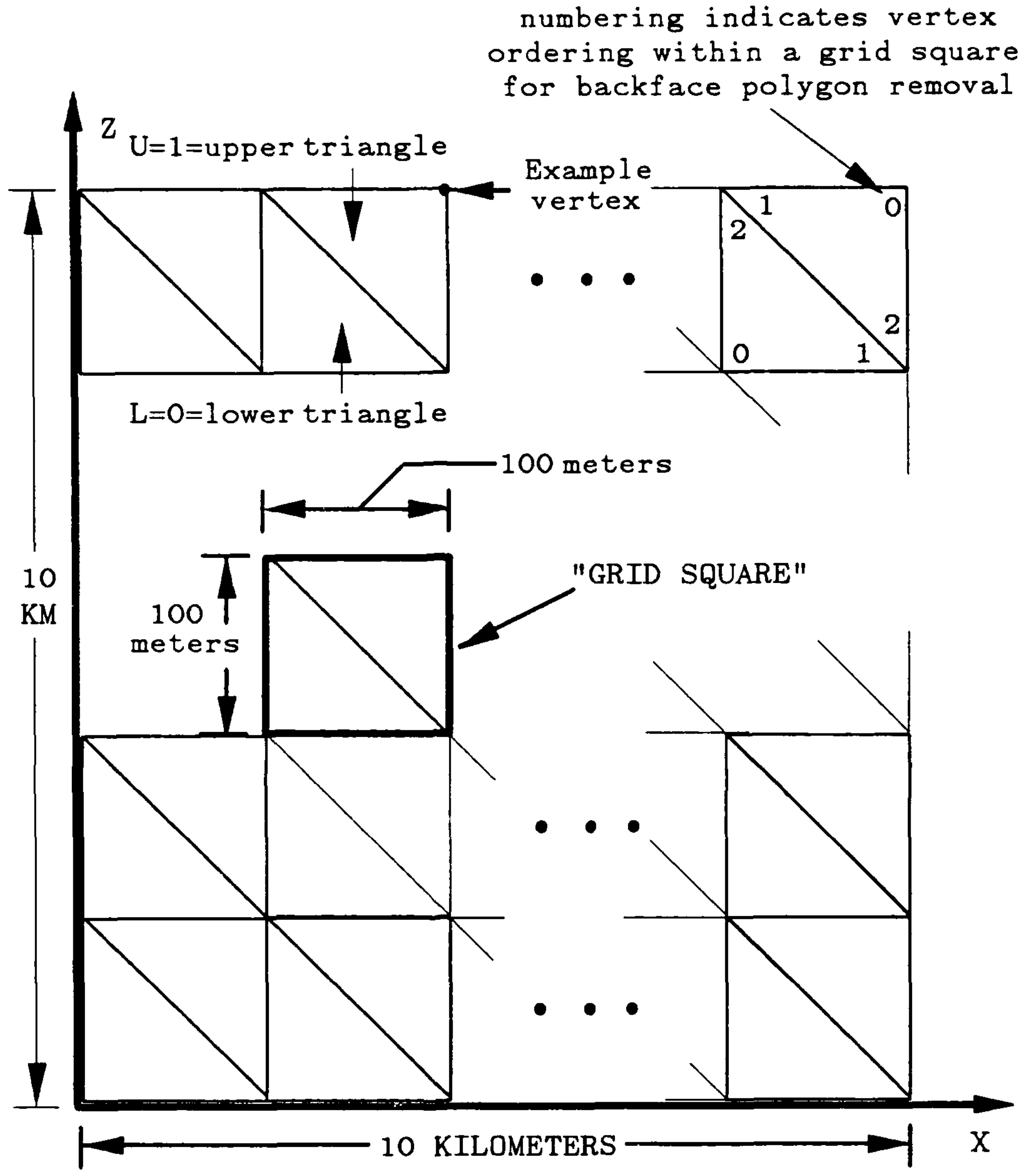


Figure 3.1 Terrain Polygons [Ref. 6]

C. VEHICLE CHARACTERISTICS

The following section describes the requirements for vehicle dynamics and terrain traversing characteristics necessary for model validations. Although the FMC autopilot uses a M113A1 as its primary testbed, the authors' decided to focus on a different vehicle to enhance the scope of research and produce a system in which weapon characteristics could later be included.

1. The Vehicles

The tracked vehicle modeled in this thesis is the M1A1 Abrams Tank, which is the Army's primary ground combat weapon system. It was designed to use mobility and firepower to close and destroy enemy forces. Deliveries of this vehicle began in August 1985. It is equipped with a 120mm gun with an NBC overpressure protective system.

The vehicle is 387 inches long, 143.8 inches wide and 93.5 inches high. It weighs approximately 63 tons, including a four man crew. It can operate at speeds up to 41.5 miles per hour. The main gun is capable of firing four types of cartridges: a kinetic energy Armor Piercing Fin Stabilized Discarding Sabot (APSFSDS) round, a chemical energy High Explosive Anti-Tank (HEAT) round, a Target Practice Cone Stabilized Discarding Sabot round and a Target Practice training counterpart for HEAT. The secondary armament includes one .50 caliber machine gun and two 7.62 caliber machine guns. The tank is powered by one 1500 horsepower gas turbine engine with a four speed automatic transmission. The cruising range of the vehicle is 275 miles at 29 miles per hour. It has thermal imaging sight, laser rangefinders, and a digital computer for fire control. The two main objectives of the motion simulation part of this thesis are to develop combat vehicle models that reflect realistic vehicle dynamics, without degrading the overall system performance, and to provide an interface that allows a Lisp Machine acting as the pilot to control the motion of the vehicles.

The wheeled vehicle used in the simulation as a guide vehicle is the M151A2 Army jeep. It was introduced in 1969 and is built by both The Ford Motor Company and AM General. The jeep weights approximately 2,273 pounds and has a four cylinder 141.5 cubic inch engine which develops seventy-one horsepower at 4,000 revolutions per minute. It has a four speed transmission with a synchromesh on the top three gears. A fixed differential with independent coil-spring suspension all around makes the M151A2 much safer than its predecessors which had both steering and stability problems.

2. <u>The Vehicle Simulation</u>

For the purpose of this study, it is necessary for the LISP Machine to be able to control vehicle courses, speeds, and vision perspective in order to adequately represent capabilities of the FMC automatic pilot.

The course changes can be relative (turn right 20 degrees) or real (come to course 320). The speed changes can also be relative (increase speed by 5 mph) or real (come to speed 29 mph). The C code used to implement the course changes are shown in Appendices P and Q. Perspective is the out-the-window view from a designated vehicle. The simulation allows the Lisp Machine to select a vehicle and then display changes to reflect the view from the new vehicle. Course, speed, and position information regarding each of the vehicles in the simulation is communicated continuously to the LISP Machines.

The mathematical model used to describe the behavior of the tank and jeep is derived from that of [Ref. 2], As in Tan's work, the notation used in this study follows the notation adopted by Frank and McGhee [Ref. 27] as closely as possible. The vehicle is confined to a flat surface. This simplifies the model by making $Z = 0$ and $\dot{Z} = 0$, and collapsing the position vector to a two dimensional vector. The vehicle is idealized to a *point mass* by ignoring the rotational moment of inertia.

3. Steering Model

a. Jeep

A further simplification of the jeep model is achieved by assuming that the vehicle turning rate, $\dot{\psi}_{jeep}$, is linearly proportional to both the forward velocity and the steering wheel angle, θ. That is:

$$\dot{\psi}_{jeep} = k_{\dot{\psi}_{jeep}} \theta \dot{x}, \quad k_{\dot{\psi}jeep} > 0 \tag{3.2}$$

To calculate the associated turning radius, R_{jeep}, note that the time to rotate the vehicle through an angle 2π is:

$$t_{jeep_{2\pi}} = \frac{2\pi}{|\dot{\psi}_{jeep}|} = \frac{2\pi}{k_{\dot{\psi}_{jeep}} |\theta \dot{x}|} \tag{3.3}$$

while the distance traveled is

$$d_{jeep} = 2\pi R_{jeep} = |\dot{x}| t_{jeep_{2\pi}} \tag{3.4}$$

Thus

$$R_{jeep} = \frac{|\dot{x}|}{2\pi} t_{jeep_{2\pi}} = \frac{|\dot{x}|}{k_{\dot{\psi}_{jeep}} |\theta \dot{x}|} \tag{3.5}$$

Or

$$k_{\dot{\psi}_{jeep}} = \frac{\frac{1}{R_{jeep}}}{|\theta|} \tag{3.6}$$

This equation shows that tight steering is reflected by large values of $k\dot{\psi}_{jeep}$, and loose steering is reflected by small values of $k\dot{\psi}_{jeep}$.

b. Tank

A simplification of the tank model is achieved by assuming that the vehicle turning rate, $\dot{\psi}_{tank}$, is linearly proportional to the steering wheel angle, θ. That is,

$$\dot{\psi}_{tank} = k_{\dot{\psi}_{tank}}\theta,\ k_{\dot{\psi}_{tank}} > 0 \tag{3.7}$$

Thus, since

$$\dot{x} = \dot{\psi}R_{tank} \tag{3.8}$$

it follows that

$$R_{tank} = \frac{\dot{x}}{\dot{\psi}_{tank}} = \frac{\dot{x}}{k_{\dot{\theta}}|\theta|} \tag{3.9}$$

This relationship reflects the ability of a tank to turn in place when its forward speed is zero.

4. Longitudinal Control Model

Longitudinal acceleration control for both the tank and jeep can be approximated by

$$\ddot{x} = \frac{1}{\tau_a}(\dot{x}_c - \dot{x}) \tag{3.10}$$

where $\dot{x}_c$ is the commanded velocity, which in turn is a function of the accelerator position, and τ_a is the acceleration time constant. A step change in $\dot{x}_c$ at $t = t_0$ produces a velocity profile of

$$\dot{x}(t) = \dot{x}(t_0) + (\dot{x}_c(t_0) - \dot{x}(t_0))e^{-\frac{t-t_0}{\tau_a}} \tag{3.11}$$

Combining all of the above equations results in the state vector

$$\vec{x} = (x_E, y_E, \dot{x}, \psi)^T \tag{3.12}$$

which is suitable for either a jeep r a tank. If the control vector, $\vec{u}$, provided by the human operator is defined as

$$\vec{u} = (\dot{x}_c, \theta)^T \tag{3.13}$$

then, from the above analysis, the component form of the state equation for the jeep is:

$$\dot{x}(1) = \dot{x}_E = \dot{x}\cos\psi = x(3)\cos x(4) \tag{3.14}$$

$$\dot{x}(2) = \dot{y}_E = \dot{x}\sin\psi = x(3)\sin x(4) \tag{3.15}$$

$$\dot{x}(3) = \ddot{x} = -\frac{1}{\tau_a}x(3) + \frac{1}{\tau_a}u(1) \tag{3.16}$$

$$\dot{x}(4) = \dot{\psi} = k_{\dot{\psi}}\dot{x}(3)\,u(2) \tag{3.17}$$

and the component form of the state equation for the tank is:

$$\dot{x}(1) = \dot{x}_E = \dot{x}\cos\psi = x(3)\cos x(4) \tag{3.18}$$

$$\dot{x}(2) = \dot{y}_E = \dot{x}\sin\psi = x(3)\sin x(4) \tag{3.19}$$

$$\dot{x}(3) = \ddot{x} = -\frac{1}{\tau_a}x(3) + \frac{1}{\tau_a}u(1) \tag{3.20}$$

$$\dot{x}(4) = \dot{\psi} = k_{\dot{\psi}}u(2) \tag{3.21}$$

For manual control, $\vec{u}(t)$ is provided by the human operator.

5. Vehicle Bounce Model

With the above equation, the vehicle moves smoothly over the planar patches of terrain with no vertical motion of any kind. This is certainly not appropriate to simulation of off-road locomotion. To render the simulation more realistic, a *bounce* angle representing vehicle pitch excursions is added to the geometrically determined pitch angle. The equation used for this purpose is

$$Bounceangle = \alpha_{bounce} * bounceangle + k_{bounce} * randomnumber \tag{3.22}$$

where the random number is uniform from -1 to +1. The values for α_{bounce} and k_{bounce} are adjusted experimentally to give realistic behavior and are, in general, dependent on both speed and terrain roughness. However, to achieve a stationary random process, the value for bounce is confined to the open interval (0,1).

D. THE DIVISION BETWEEN GRAPHICS AND LOGICAL ASPECTS OF THE PROBLEM

This section describes the functional requirements for the prototype simulation system and describes the organization and division of labor used to develop its structure.

1. Tactical Assessments

The simulation system developed in this thesis possesses a rudimentary tactical assessment capability to analyze the following factors: mission, enemy forces, terrain and weather, assets, and support available to the system. The analysis of these factors are necessary to develop plans and execute tasks to enable the system to accomplish its tactical mission. The tactical assessment capability is not currently integrated into the prototype system. Rather, inputs are entered by keyboard to a Prolog program residing on a Vax 780 computer in an interactive session. The plan is produced and output to a terminal. The plan is then communicated to the command and control system on the LISP machine via keyboard.

The prototype tactical assessment capability, implemented in Prolog, has been developed using first order predicate calculus and forward chaining as the deductive inference engine. With first order predicate calculus it is quite easy to produce the symbolic information and rules upon which to draw inferences and make deductions. Forward chaining is used in the prototype because the analysis is quite open ended. All tasks and actions to be performed based upon the factors used in the analysis are not known before hand. Backward chaining is then used to query the task list to create a viable plan of action for the command and control subsystem.

The plan of action consists of determining the formation for the unit to assume and the method of attack. The formations that can be determined are *column*, and *line*. The column formation is determined if the assessment indicates that speed and control are required, fire and maneuver to the flanks optimal to address a threat, and that vision and maneuver are restricted. The line formation is indicated for crossing open areas or in the assault where fire and maneuver are necessary to the front. The prototype system currently is able to assume the column/file and line formations.

The method of attack is determined from one of two choices, a frontal assault or a single envelopment. The frontal assault is determined if fire superiority can be gained, there are no key terrain features that would afford the establishment of a base of fire for an envelopment, and there is no covered and concealed avenue of approach available to a maneuver element. An envelopment is determined if there is an acceptable avenue of approach and key terrain necessary for the establishment of a base of fire. Currently the prototype system is capable of conducting the movement coordination necessary for the frontal assault.

2. Autonomous Vehicles

Simulated tanks with the control characteristics similar to those of the existing FMC Autonomous Land Vehicle are modeled. The model is conceptually organized in two distinct functional parts: its graphics instantiation, with vehicle controller functions on the IRIS, and its rule based, expert system behaviors implemented on the Lisp Machines. The tanks act autonomously in much the same manner as the FMC vehicle. Specifically, each tank possesses a simulated vision capability, a pilot, and the ability to send vehicle steer and reference velocity commands to a vehicle controller. The tank performs functionally according to the algorithm presented in Figure 3.2.

The pilot possesses additional capabilities, besides those being developed at FMC [Ref. 8], to allow the vehicle to act as an integral part of a tactical autonomous unit. The additional capability allows the tanks to perform the following: station keeping, communication, diagnostic information and isolation of behavior.

```
Loop
  Check for commands from the Command and Control Subsystem.
      If change in formation, acquire rules and facts
      necessary from disk storage and implement.

  Perform a visual scan of the environment.

  For each objects identified:

      Establish its position in reference to the
        tank's body coordinate system.

      Approximate its future location at beginning
        of next iteration of the algorithm.

      Produce low level observations about the
        object as input to the taskgenerator.

  EndFor

  Generate tasks in the taskgenerator using the low level
    observations and knowledge and rules necessary to
    complete currently assigned goals.

  Display diagnostic information and explanations for each
    task generated.

  Execute communications tasks to Command and Control subsystem.

  Execute tasks generated by communicating sequences
    of vehicle steer and reference velocity commands
    to the vehicle controller residing on the IRIS.

EndLoop.
```

Figure 3.2 Tank Algorithm

Station Keeping. Based upon commands sent to the vehicle by the lead vehicle, the vehicle assumes its designated place in a tactical formation and keeps its station with the formation until further commanded. Currently the tanks use three sets of simple rules that allow the vehicles to assume a line, column, or file formation. For each formation, each tank possesses knowledge about who it is, the type of formation, its guide vehicle, and the vehicles that should be to its flanks, front and rear. Rules for each formation are divided into four functional categories: collision avoidance, speed determination, direction determination, and stationing. These rules are presented in Figures 3.3 through 3.6.

Communication. Bi-directional inter-vehicular communication is simulated to allow unit communication and control. This is necessary for signaling formation changes, halts, etc., to control the unit as it moves through different phases of the mission. Communication is initiated by the command and control subsystem.

Diagnostic Information. To measure performance and validate concepts as the research progresses, each vehicle produces a set of information displayed upon the IRIS and the LISP Machine as to what courses of action were available to it based upon its simulated environment. The vehicle explains each task it generates and executes in terms of what it perceived in its vision phase and what rules it applied to those observations.

Isolation of Behavior. Each tank is represented upon a separate computer to provide isolation of observable phenomena within the command and control system.

3. Command and Control

A high level command and control Subsystem is simulated upon a LISP Machine and the VAX that provides centralized autonomous command and control functions to the individual tanks and acts as a single interface to the autonomous vehicles in the unit. This allows an isolation of observable phenomena for the tactical assessment

```
Avoid Collision To The Right:
    If
      the vehicle is or will be too close to an object, and
      the object is to the right of the vehicle,
    Then
      move to the left.

Avoid Collision To The Left:
    If
      the vehicle is or will be too close to an object, and
      the object is to the left of the vehicle,
    Then
      move to the right.

Avoid Collision Ahead:
    If
      the vehicle is or will be too close to an object, and
      the object is ahead of the vehicle,
    Then
      If
        not enough time to maneuver,
      Then
        Stop.
      ElseIf
        able to maneuver,
      Then
        maneuver around object in flank
        with greatest maneuver room.

Avoid Collision From Behind:
    If
      the vehicle is or will be too close to an object, and
      the object is behind the vehicle and closing,
    Then
      match the object's speed.
```

Figure 3.3 Collision Avoidance Rules

```
Change Speed:

    If
      vehicle is on course with its guide vehicle, and
      vehicle is behind or ahead of its station,

    Then
      change speed to move vehicle to position by
      next iteration of tank algorithm.


Match Speed:

    If
      vehicle is on course with its guide vehicle, and
      vehicle is on station with its guide vehicle,

    Then
      match speed of the guide vehicle.


Stop:

    If
      guide vehicle is stopped, and
      vehicle on station with guide vehicle,

    Then
      stop vehicle on station.
```

Figure 3.4 Speed Determination Rules

```
Turn Left:
    If
      vehicle is off course from its guide vehicle and
      relative right to the direction of guide vehicle
      course,
    Then
      turn left the angular difference to come about.

Turn Right:
    If
      vehicle is off course from its guide vehicle and
      relative left to the direction of guide vehicle
      course,
    Then
      turn right the angular difference to come about.
```

Figure 3.5 Direction Determination Rules

function as well as centralizing the focus of one problem in the research area. The command and control subsystem algorithm is presented in Figure 3.7.

4. Communication

Communication on two levels is simulated for the prototype system: terrain to virtual tank and virtual tank to virtual tank. Terrain to virtual tank provides the simulated physical interaction of the tank with its environment. Tank to tank communication provides simulation of unit size command and control interaction.

Terrain to virtual tank simulates the virtual tank's vision capability from the IRIS computer upon which the graphics system is represented to the autonomous vehicles expert system located upon the LISP machines. The LISP machines simulate the FMC pilot to vehicle controller function [Ref. 1] by communicating to the IRIS reference velocity and course information. The IRIS, acting as the vehicle controller, moves the tank's graphical instantiation over the representation of the terrain model based on LISP

```
Close Right With Guide:
    If
      vehicle is too far from guide, and
      vehicle is left of guide, and
      guide vehicle is normally vehicle's right vehicle,
    Then
      move to the right.

Close Left With Guide
    If
      vehicle is too far from guide, and
      vehicle is right of guide, and
      guide vehicle is normally vehicle's left vehicle,
    Then
      move to the left.

Assume Correct Position in Relation to Guide:
    If
      vehicle is on course with guide, and
      vehicle is left/right of guide,
      but vehicle should be right/left of guide,
    Then
      drop behind guide,
      turn 90 degrees right/left,
      proceed until past guide,
      turn 90 degrees left/right.
```

Figure 3.6 Stationing Rules

machine commands. Additionally, the IRIS provides information concerning other tanks upon the terrain model as well as terrain information.

Virtual tank to virtual tank simulates the communications characteristics of tactical units in command and control functions such as commands and information. The virtual commander communicates with the battlefield in much the same was as tanks do to provide it with the information it requires to make tactical assessments and formulate and carry out plans.

```
Loop
   Check for command mission interruption.
        If change in mission:
             Acquire METT information.
             Establish and queue position of:
                 assembly area,
                 line of departure,
                 march control points,
                 final coordination line,
                 objective location.

   Perform a visual and map scan of the environment from lead vehicle.

   Perform tactical assessment.

   For current position in the queue:

        Establish its position in reference to the
          unit's body coordinate system.

        Approximate its future location at beginning
          of next iteration of the algorithm.

        Produce low level observations about the
          position as input to the taskgenerator.

   EndFor

   Execute communications tasks to subordinate vehicles.

   Execute Tank Algorithm if Command and Control is centralized in
        one tank.

EndLoop.
```

Figure 3.7 Command and Control Algorithm

Because of various computer architectures available at the Naval Postgraduate School, an application medium has been developed over the communication protocols used that is conceptually similar on each machine. The protocol chosen for battlefield to tank/commander was TCP-IP reliable mode [Ref. 26]. This was felt to be necessary to prevent messages to the terrain model on the IRIS from becoming lost or out of sequence. The protocol chosen for tank to tank was Chaos [Ref. 26]. It was felt that it was faster yet reliable enough to communicate over the relatively short distances encountered in the lab. Additionally, scripts could be developed for the system that could address faulty communication during operations.

E. SUMMARY

This chapter provides a detailed discussion of the problems considered for this study. It identifies the major requirements of a prototype system to effect this research organized into three main areas: terrain model, vehicle characteristics, and the division between the graphical and logical aspects of the problem. The terrain model describes the characteristics of the environment simulated as a test bed for the command and control algorithms. The vehicle characteristics section is used to describe the general characteristics of the military vehicles used in the Autonomous Vehicle Motion Simulator and the basic commands available to the LISP machine for motion control of the vehicles over the terrain. Algorithms are presented that are used to model the behavior of individual tanks and the command and control subsystem.

IV. STRUCTURE OF PROTOTYPE SIMULATION SYSTEM

A. INTRODUCTION

The autonomous vehicle simulation is a system utilizing four different computer architectures, three languages, four networking packages, and an expert system shell. The simulation software is currently organized into five major functional areas: the graphics simulation, the vehicle simulation, the command and control modules, the tank to battlefield communication, and the command and control to tank communication. The four computer architectures utilized are the Symbolics 3600 line of LISP Machines, the Texas Instrument Explorer LISP Machine, the IRIS Graphics Workstation, and the Vax 11/780. The languages implementing the system are Prolog, C, and Lisp with Flavor extensions. The expert system shell utilized is the KEE expert system. A discussion of the software developed for the prototype follows.

B. TERRAIN AND VEHICLE SIMULATION

The autonomous vehicle simulator models the motion of remotely piloted vehicles, such as jeeps, tanks, or trucks, one of which is designated the *driven* vehicle. The *driven* vehicle models a vehicle with an on-board video camera capable of transmitting live pictures of the battlefield to a distant operator's console. The simulator displays a real-time, dynamic, three dimensional, out-the-window view (a driver's view perspective) of the terrain and other vehicles (Figure 4.1). An interactive user interface and a two-dimensional contour map display allow the operator to define each vehicle to be used in the simulation. The initial vehicle locations, courses, speeds, and the selection of a driven vehicle are determined via this interface.

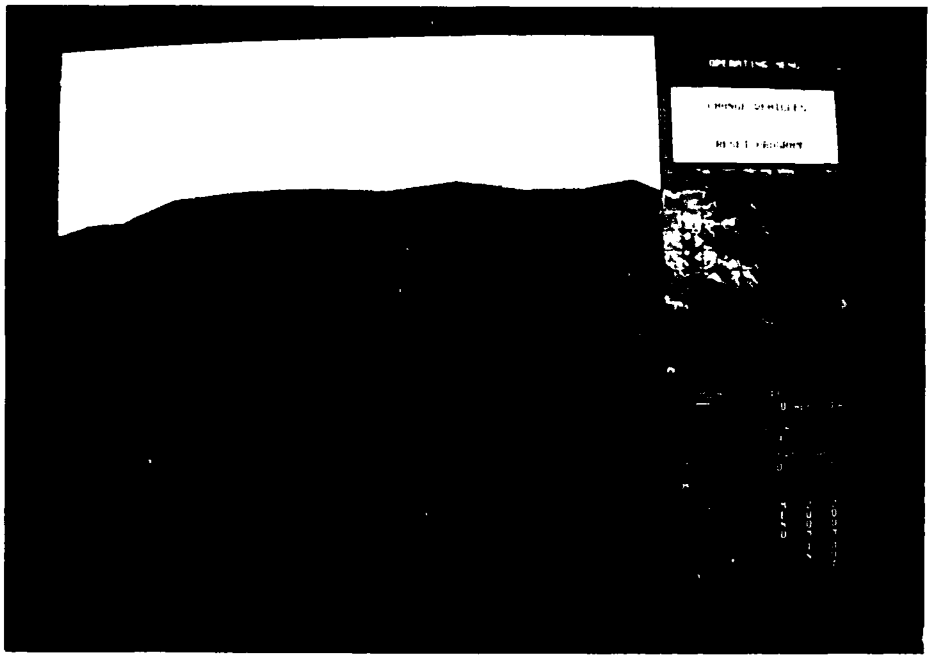


Figure 4.1 Out-The-Window View

Once the simulation begins, a three-dimensional view of the terrain, obtained from a terrain database provided by the Defense Mapping Agency, is displayed. The operator can interactively control the motion of the vehicle designated as the driven vehicle.

The operator controls the driven vehicle's course, speed, steering angle, driver tilt, and line of sight look direction, by the knobs on the dial box. The apparent viewing volume of the driven vehicle can be controlled by the mouse. The field of view changes plus or minus five degrees for every cycle of the display loop that the mouse button is depressed.

1. Terrain

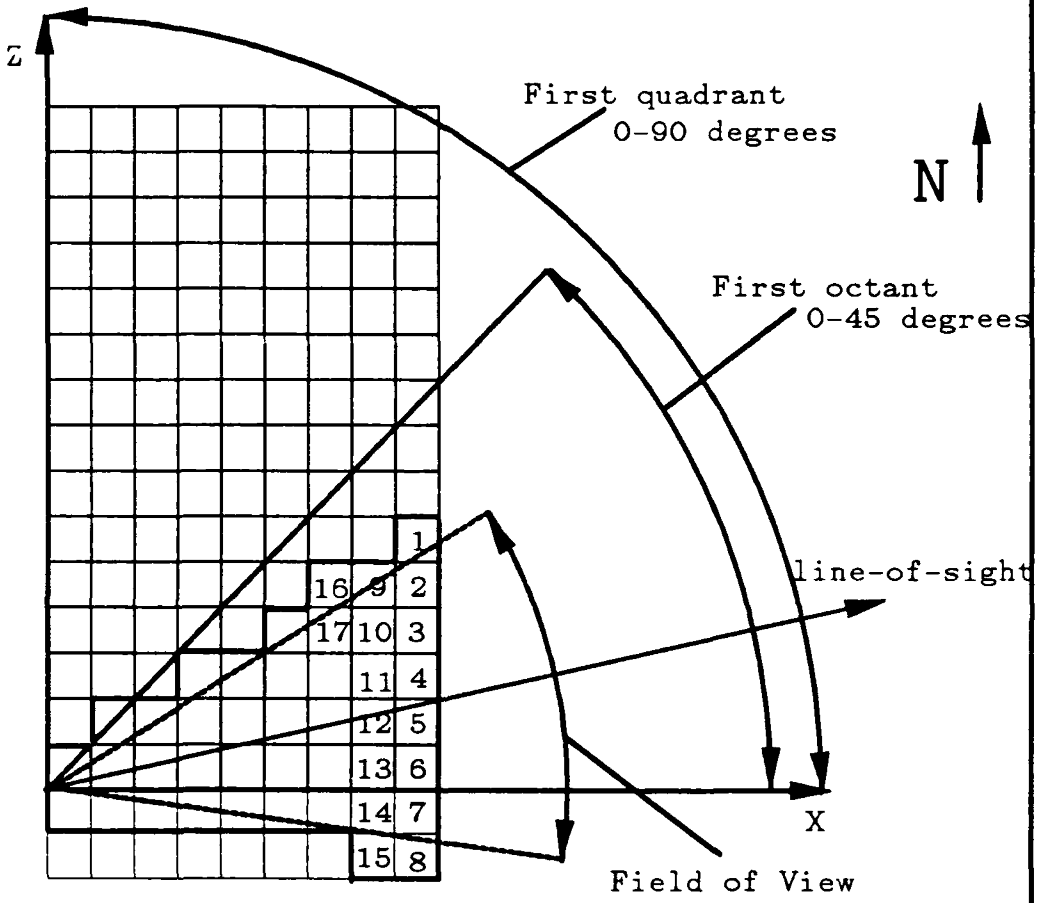
The simulator uses a digital terrain elevation database provided by the Defense Mapping Agency (DMA) to draw the three-dimensional scene. The terrain model is described in Chapter 3.

2. Hidden Surface Elimination

Hidden surface elimination is accomplished by a real-time implementation of the Painter's Algorithm. The Painter's Algorithm simply draws objects in the scene in depth sorted (furthest to nearest) order [Ref. 22]. For drawing the terrain, the correct polygon drawing order for hidden surface elimination is a function of the driver's field of view from the vehicle currently being operated (Figure 4.2). The least number of grid squares are drawn by partitioning the viewing area into octants. The order that each grid square within an octant is drawn in, from furthest to nearest, is based on a scan line algorithm (Figure 4.3(a)). If the field of view is in the eighth octant, the scan lines are defined by indices *startx* and *startz*. *Startz* is incremented until a *stopz* is reached. Before *startz* is incremented, all vehicles located in the grid square that was just drawn are also drawn. One vertical scan line is shown in (Figure 4.2(b)). The next scan line is drawn by moving the *startx* one position closer to the viewer and repeating the process. This process is repeated until all grid squares in the octant are drawn.

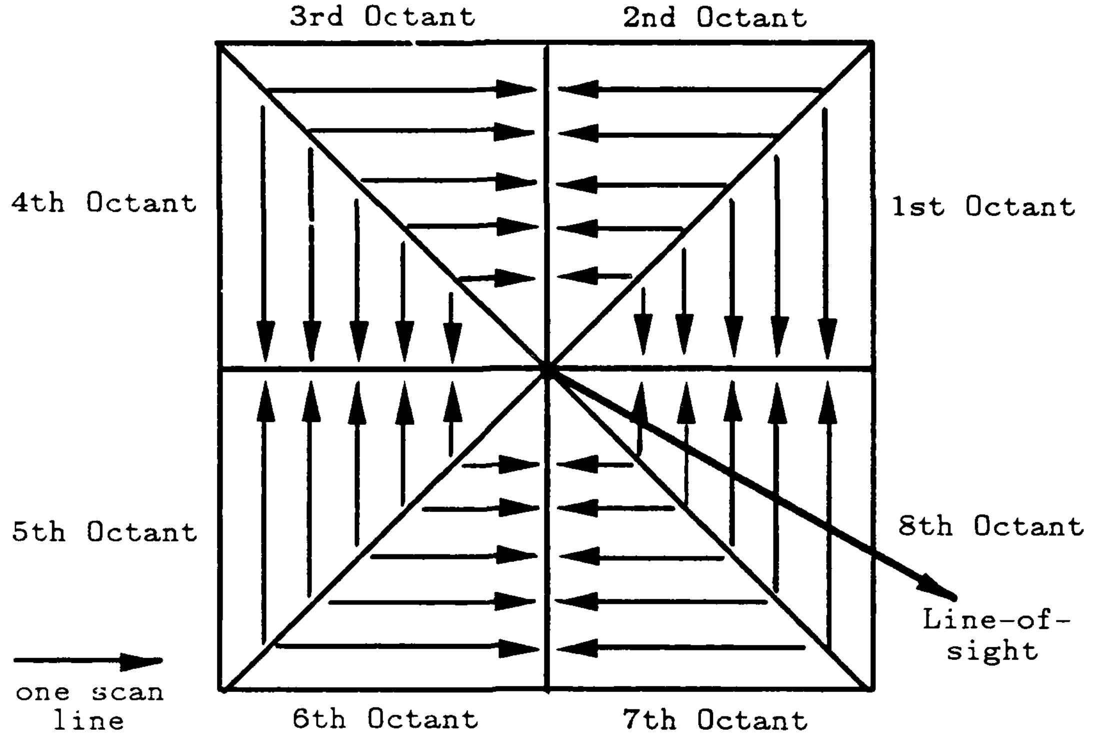
After the entire scene is drawn, the vehicles in the viewer's grid square are drawn again. This is one way to ensure vehicles drawn in adjacent grid squares are painted over by vehicles in the viewer's grid square [Ref. 6].

Vehicles located in the center of a grid square are drawn immediately after the grid square that they occupy is drawn. Vehicles crossing grid square boundaries are drawn only once. The grid square that they are drawn in is determined by the quadrant they are in and which boundary the vehicle is crossing. A vehicle is drawn in an adjacent

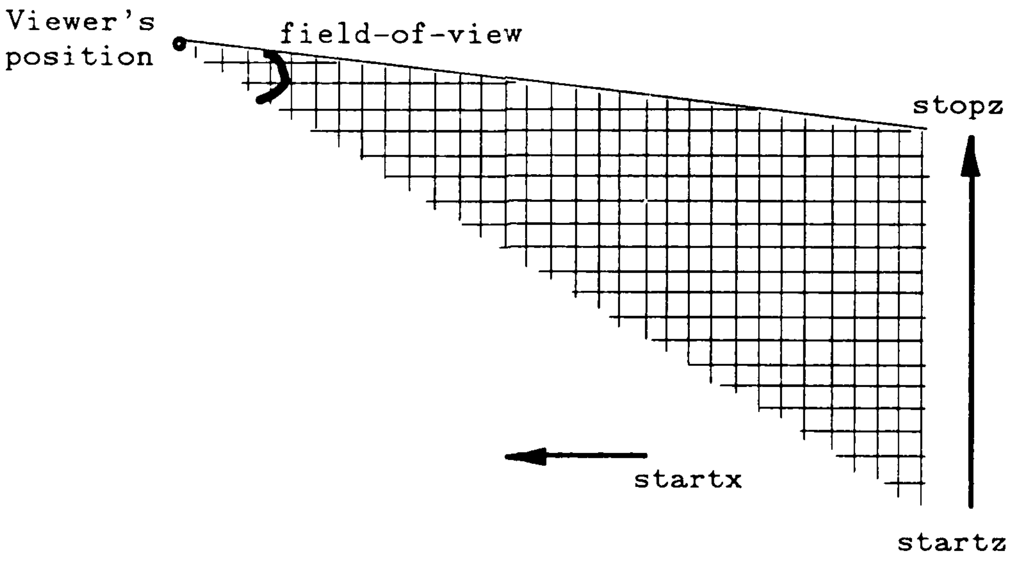


numbers indicate drawing order
north to south, east to west

Figure 4.2 First Quadrant Drawing Order Example [Ref. 6]



a.



b.

Figure 4.3 Octant Scan Lines [Ref. 6]

grid square only if it is near certain edges. The edges are determined by the order of the Painter's Algorithm and are shown in Table 1 [Ref. 6].

In Figure 4.4, the line-of-sight from the driven vehicle 'A' is in quadrant one. With this line-of-sight, vehicles near a southern or eastern grid square edge are drawn after the adjacent grid square in that direction rather than in the grid square the vehicles occupy. Vehicle 'B' in Figure 4.4 is located at the southern edge of grid square three. Since the Painter's Algorithm draws grid square three before grid square four, the part of the vehicle overlapping grid square four would be painted over by grid square four if the vehicle is drawn in grid square three. To correctly draw the vehicle and both grid squares it overlaps, the vehicle must be drawn after grid square four.

3. Vehicles

The vehicles are created as graphical objects. They are constructed with the *painter's* algorithm and backface polygon removal taken into consideration for hidden surface removal [Ref. 22]. Each polygon is drawn by defining its vertices, determining its color, and then drawing the polygon using a call to a polygon fill function. All vehicles are displayed as an undistorted view of a three-dimensional, light shaded object from any viewing angle above the ground plane.

All vehicle objects (jeeps, trucks, tanks) are built during program initialization. After the objects are constructed, they are animated and oriented with respect to the

Table 1

Quadrant	Grid Square Edge
0	South, West
1	South, East
2	North, East
3	North, West

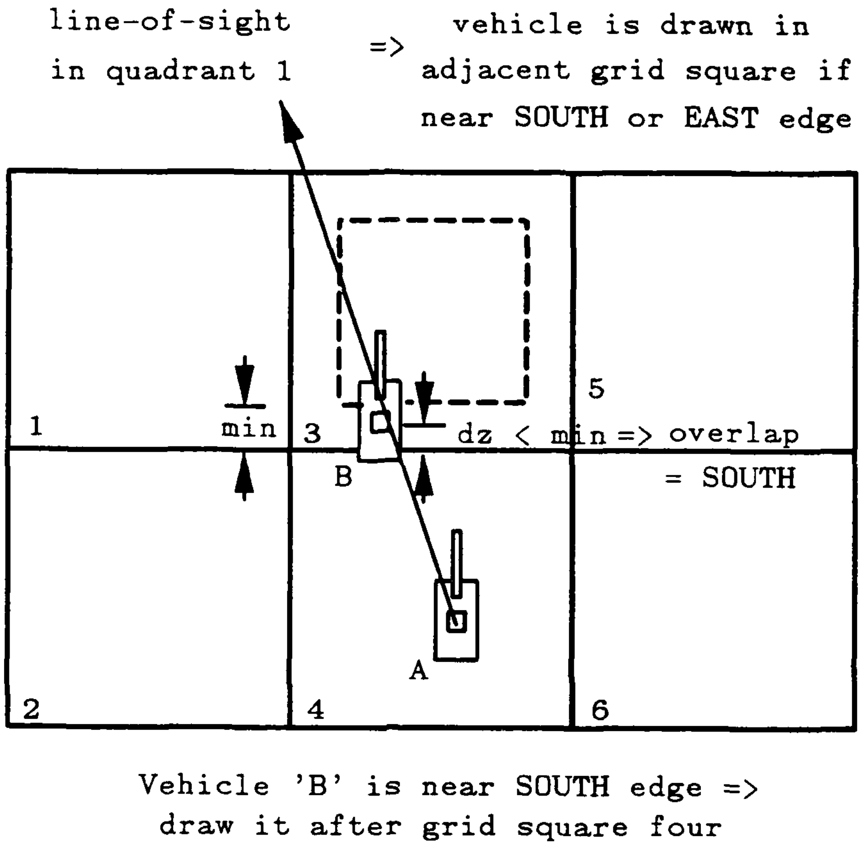


Figure 4.4 Drawing in an Adjacent Grid Square [Ref. 6]

terrain. The vehicle's course and speed are used to calculate its new position based on the distance it would have traveled in the time required to refresh the screen. Each vehicle defined is associated with an element of one of three global two-dimensional arrays. There is one array for each of the three types of vehicles. The values stored in the arrays are the integer names of the graphical objects to be drawn in each terrain grid square. All vehicles present in one grid square are associated with the same element of the array. All commands required to draw each type of vehicle are collected into the same graphical object. Vehicles are drawn by drawing the terrain grid square and then accessing the appropriate two-dimensional array to draw the vehicles that are present in that grid square.

4. Vehicle Data Structure

The simulator uses two data structures to manage the vehicle display. A linked list of vehicle definition data is created before the display loop begins and is updated with each pass through the loop. Each structure in the linked list contains all the data required to transform and orient a vehicle object to the correct position on the terrain. One object for each type of vehicle is created before the display loop begins. The drawing commands in these objects are used to draw every vehicle of that type used in the display.

The second data structure is used to manage hidden surface removal. A single two-dimensional array is used to maintain a connection between the grid squares and the order in which the vehicles present in the grid square must be drawn [Ref. 6]. Each element in the array contains a list of pointers to records in the vehicle definition list for the vehicles that should be drawn immediately after drawing the terrain grid squares. The lists are maintained in depth sorted order, from furthest to closest with respect to the *driven* vehicle. The grid square that a vehicle should be drawn in is determined by the vehicle's proximity to a grid square edge and the direction of the line-of-sight. As a result, a vehicle is drawn only once, regardless of its position on the terrain. As a vehicle overlaps a grid square, its position in the two-dimensional array changes. Figure 4.5 shows how the array changes while maintaining the linked list depth sorted order. All the functions used to draw the vehicles and terrain are performed in the display loop. Each pass through the loop represents one frame of animation. By optimizing the functions, a frame rate that simulates a real-time display is achieved.

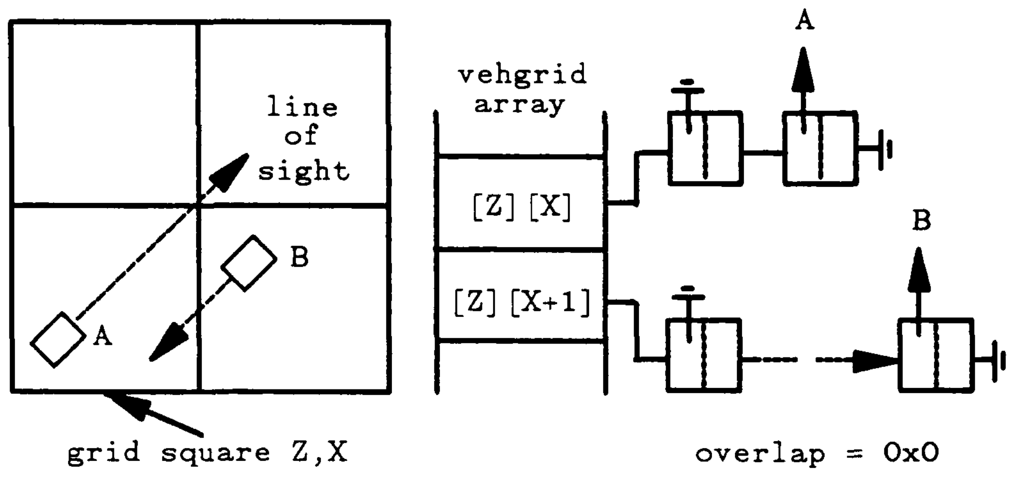
5. Manual Control Mode

There are two basic phases of the manual control mode: initialization and vehicle driving. The initialization phase provides an environment for vehicle definition and interactive input of vehicle course, speed, and position on the terrain. The driving phase provides an environment that dynamically updates the terrain displays in real-time based on operator controlled changes to the *driven* vehicle's speed, course, steering angle, and viewing volume. The operator also designates the driven vehicle.

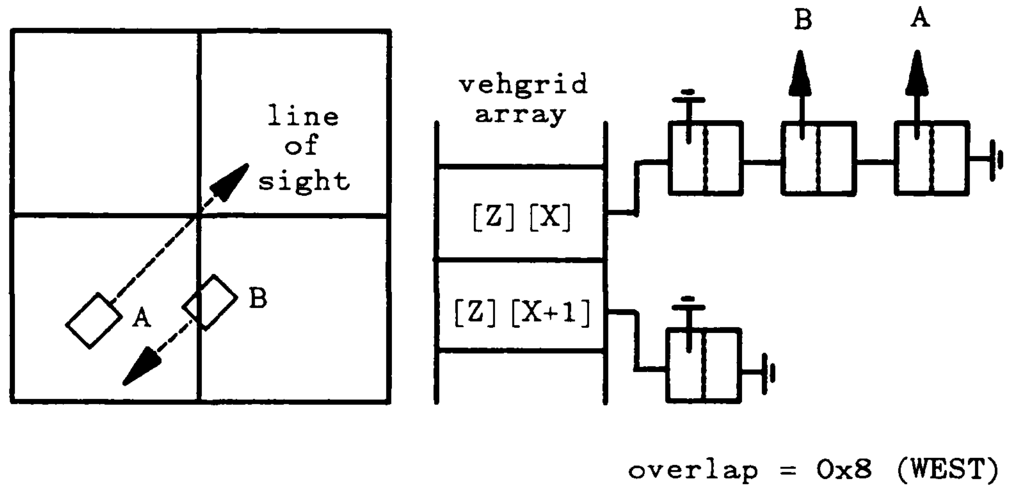
a. Initialization Phase

The initialization phase is the interactive input component of the simulator program. The display screen is partitioned into three areas as shown in Figure 4.6. A large square area (768 by 768 pixels) on the left part of the screen represents the two-dimensional contour map of the ten kilometer area over which the vehicles operate. The contours are created from the elevation data in the DMA digital terrain elevation database. The map is color coded based on the vegetation codes associated with various elevation points. The current menu is located in the upper right corner of the display. Instructions corresponding to the current menu are displayed in the lower right corner of the screen.

During this phase, the operator can define vehicles by moving the cursor on the contour map using the mouse. When the desired vehicle location on the map is selected, the coordinates are locked in by pressing the left mouse button. An iconic

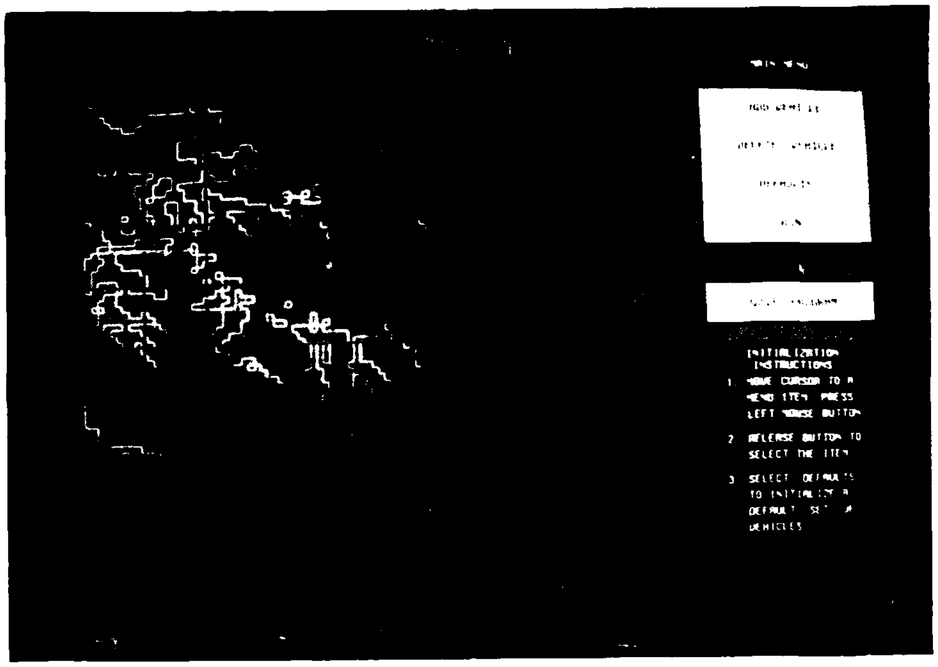


(a)



(b)

Figure 4.5 Update Vehicle Grid [Ref. 6]



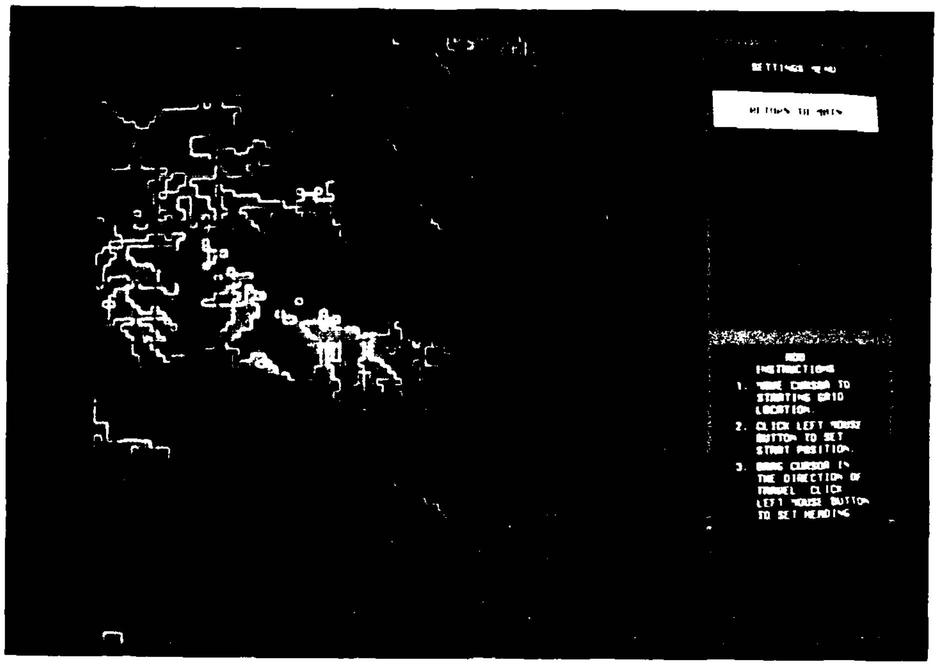
INITIALIZATION INSTRUCTIONS

1. MOVE CURSOR TO A MENU ITEM PRESS LEFT MOUSE BUTTON
2. RELEASE BUTTON TO SELECT THE ITEM
3. SELECT DEFAULTS TO INITIALIZE A DEFAULT SET OF VEHICLES

Figure 4.6 Display Screen

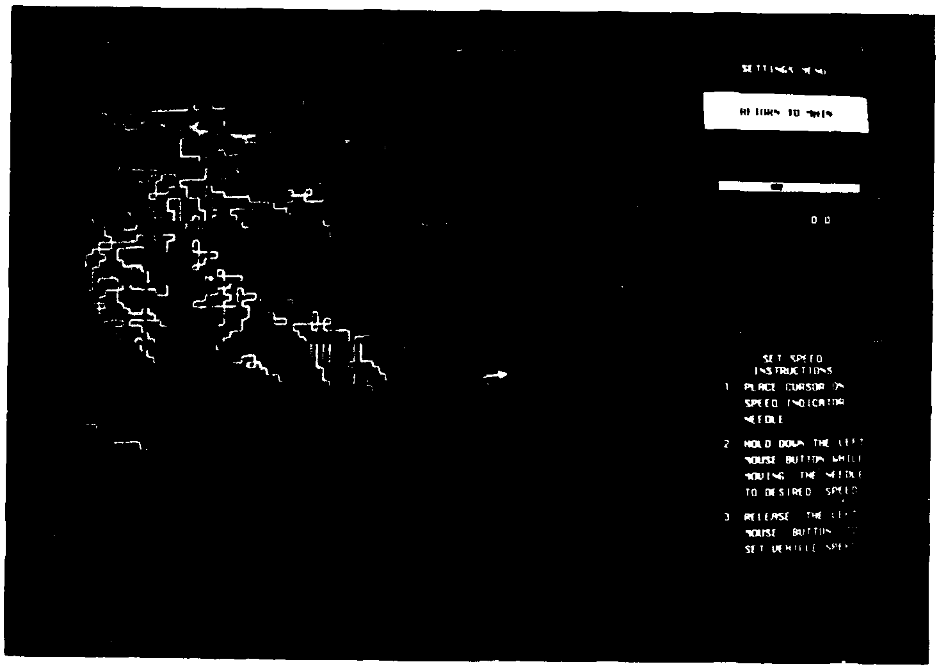
image of the vehicle appears on the map at the specified location. The operator then moves the cursor in the direction of the desired vehicle course. A rubberband line, originating at the iconic image, shows the potential vehicle course (Figure 4.7). Pressing the left mouse button locks in the course represented by the direction of the rubber-band line from the vehicle's defined location. A slider speedometer appears at this time in the menu area to allow the operator to set the vehicle's speed by moving the cursor over the desired speed and pressing the left mouse button (Figure 4.8). Once all desired vehicles have been defined, the actual simulation can begin.

The hierarchical structure of the program's main module, *veh.c*, and its major subparts are shown in Figures 4.9 through 4.12.



[illegible] INSTRUCTIONS
1. MOVE CURSOR TO STARTING GRID LOCATION.
2. CLICK LEFT MOUSE BUTTON TO SET START POSITION.
3. DRAG CURSOR IN THE DIRECTION OF TRAVEL. CLICK LEFT MOUSE BUTTON TO SET HEADING

Figure 4.7 Rubberband Line Example



SET SPEED INSTRUCTIONS
1 PLACE CURSOR ON SPEED INDICATOR NEEDLE
2 HOLD DOWN THE LEFT MOUSE BUTTON WHILE MOVING THE NEEDLE TO DESIRED SPEED
3 RELEASE THE LEFT MOUSE BUTTON TO SET VEHICLE SPEED

Figure 4.8 Slider Speedometer Example

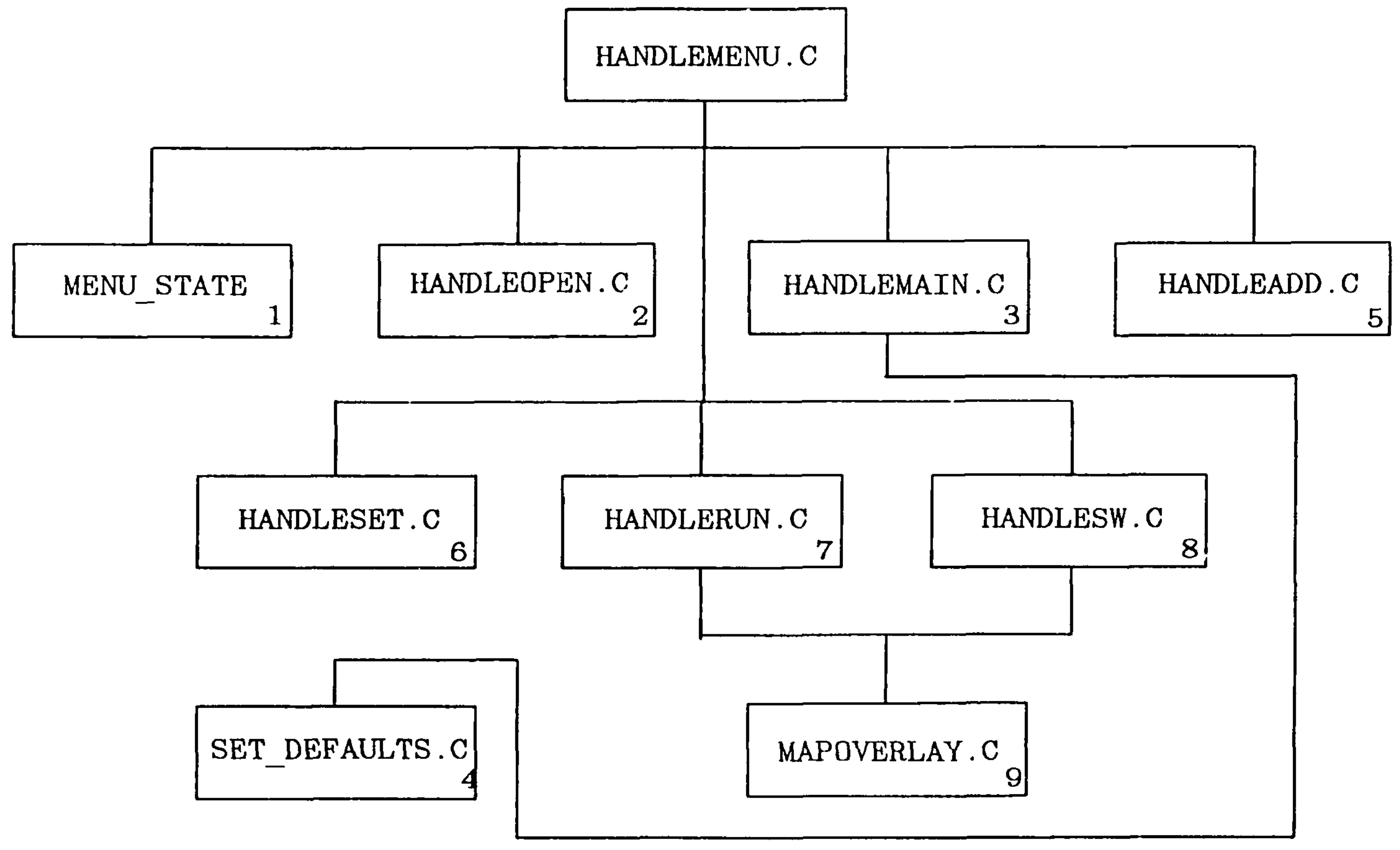


Figure 4.9 Main Module

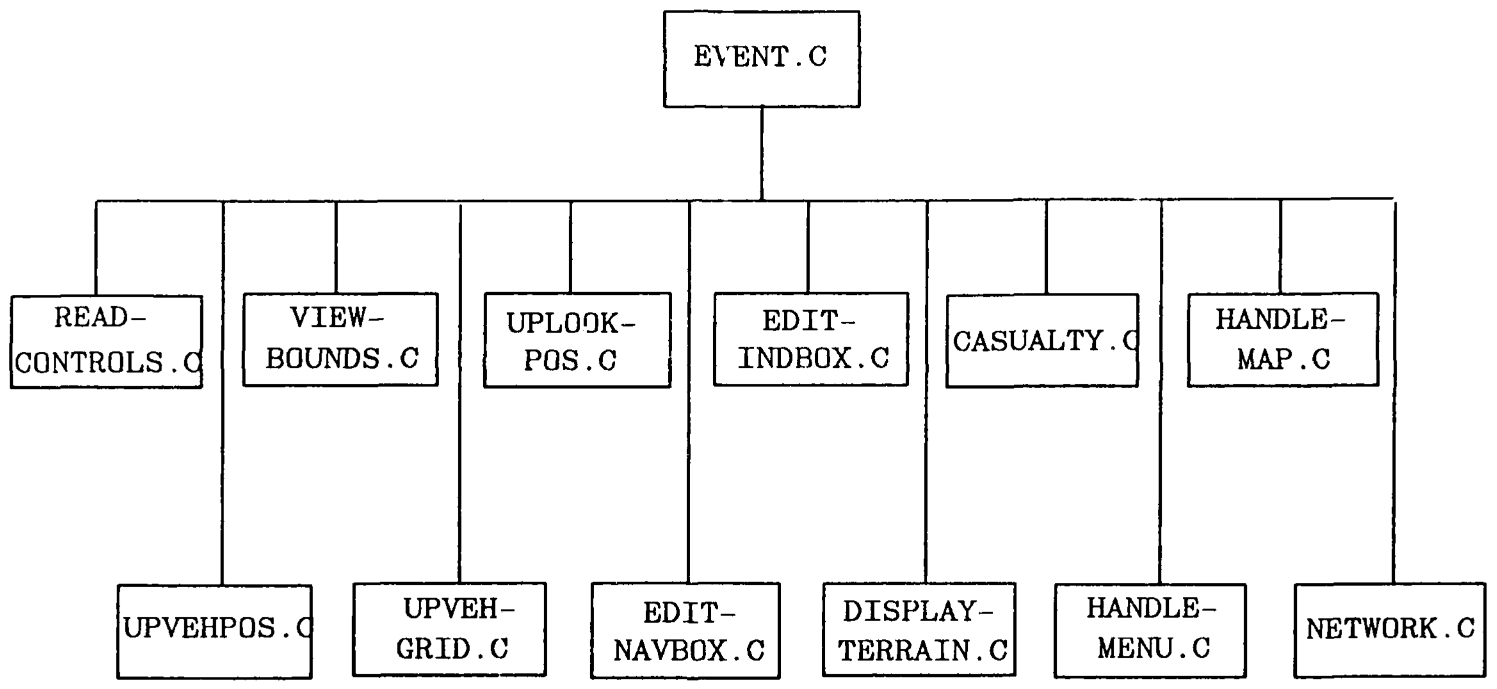


Figure 4.10 Functions Called by Event.c

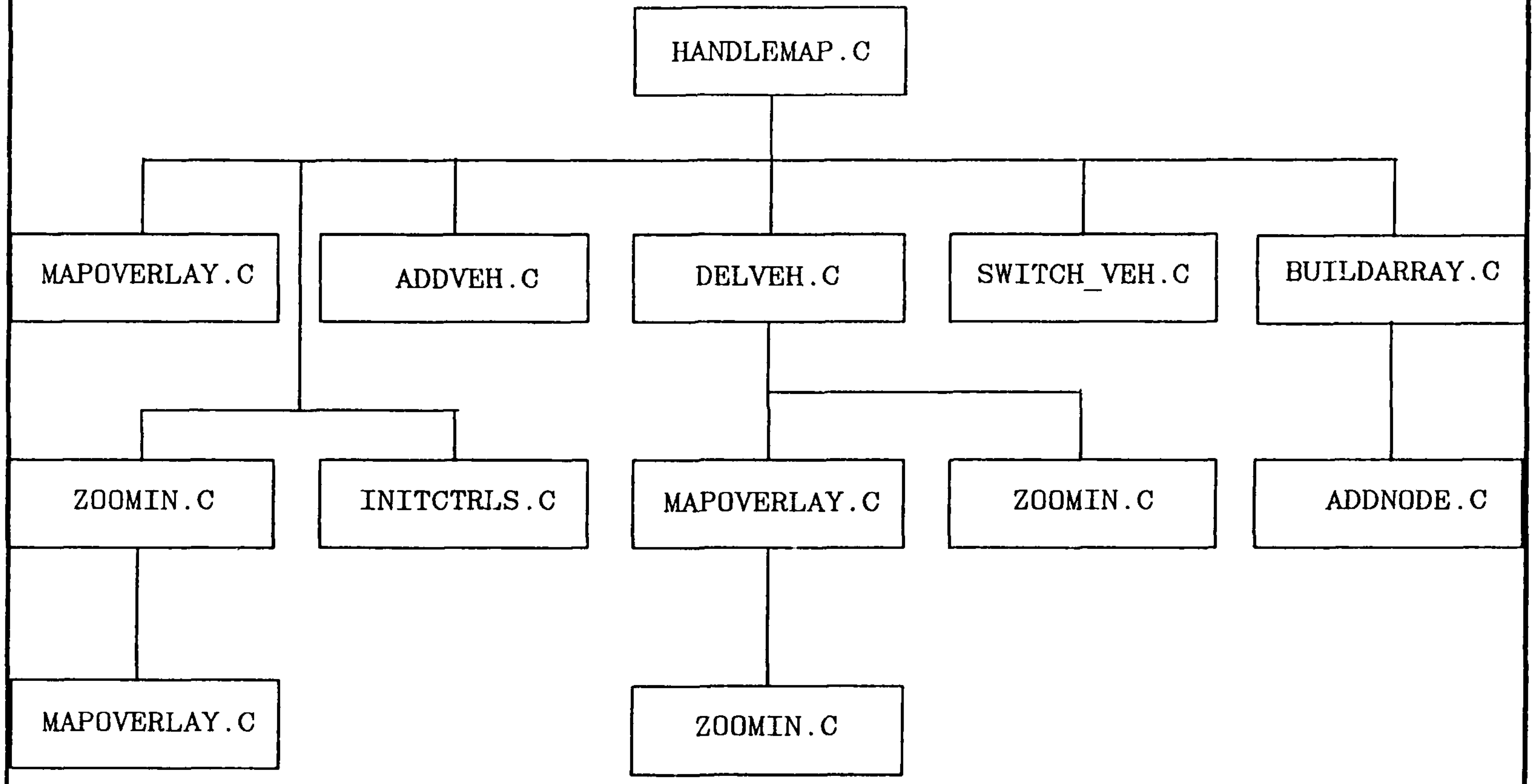


Figure 4.11 Handle Map Module

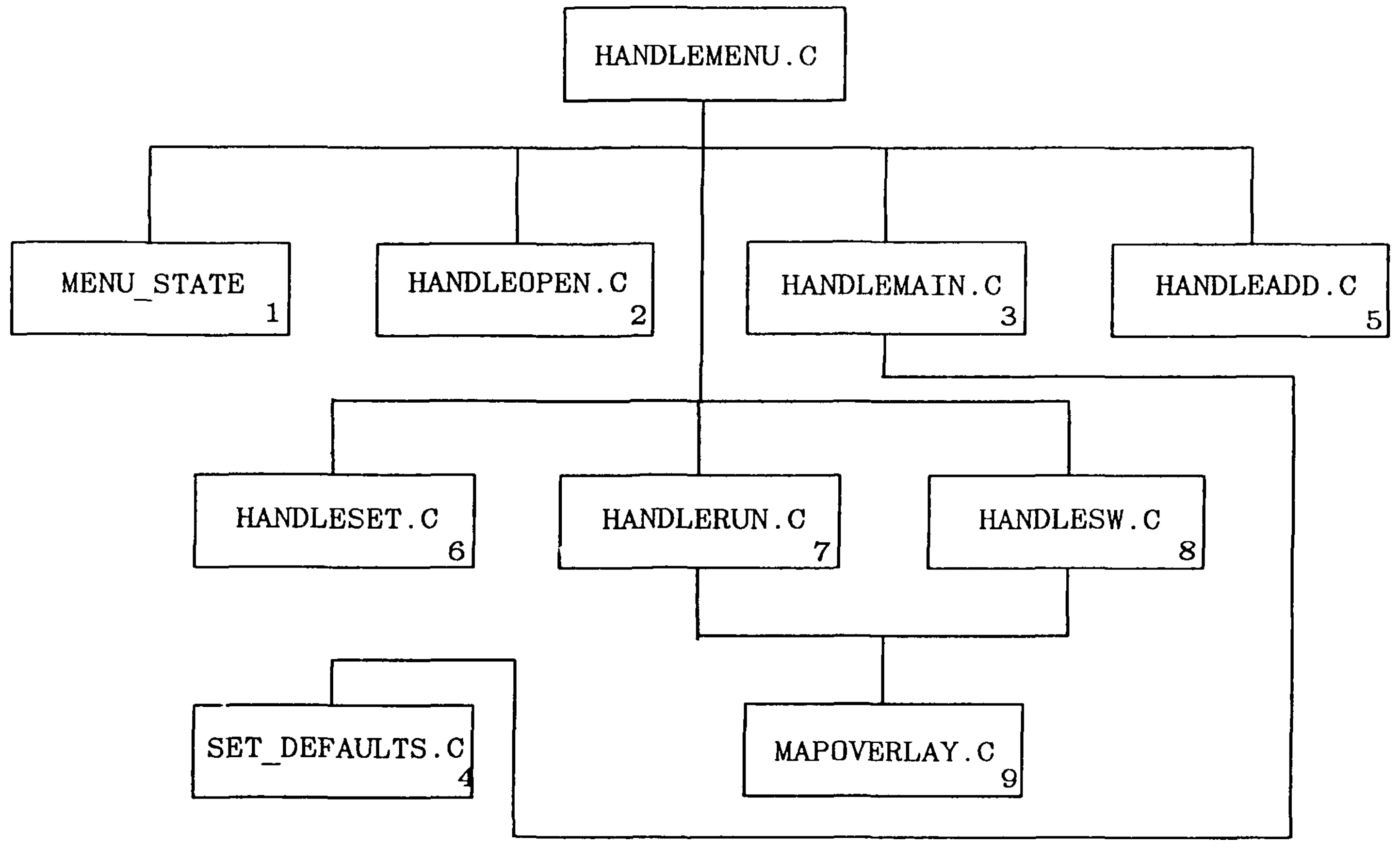


Figure 4.12 Handle Menu Module

b. Vehicle Driving Simulation Phase

The vehicle driving simulation phase provides successive real-time terrain displays to the operator as the vehicles move over the terrain. The simulation begins with the designation of a driven vehicle selected from the previously defined vehicles. The driven vehicle is selected by moving the cursor over the vehicle's iconic image on the map and then depressing the left mouse button. Selection of a vehicle starts the display loop of the simulation (Figure 4.13). The C code for the display loop driver, event.c, is shown in Appendix M.

The driving display is partitioned into four areas as shown in Figure (4.1). The large square area to the left (768 by 768 pixels) represents the out-the-window view as seen from the driven vehicle. An operating menu is displayed in the upper right side of the screen which allows the operator to change vehicles or quit the program. A contour map with the position of the driven vehicle and its viewing volume is displayed on the right, center portion of the screen. The driven vehicle's speed, view direction and available operator controls are shown in the lower right portion of the screen.

The operator is able to change the driven vehicle's speed by dialing in a new ordered speed. The vehicle accelerates/decelerates until the actual speed is equal to the ordered speed. There are two ways that the operator can change the driven vehicle's course. The first is an instantaneous course change. This is accomplished by turning the dial until the vehicle is pointing in the desired direction of travel. The second method is similar to steering an automobile. Turning the steering angle dial changes the steering wheel angle. The jeep will not turn, regardless of the steering angle, when the vehicle's speed is zero. The tank turns independently of the vehicle's speed.

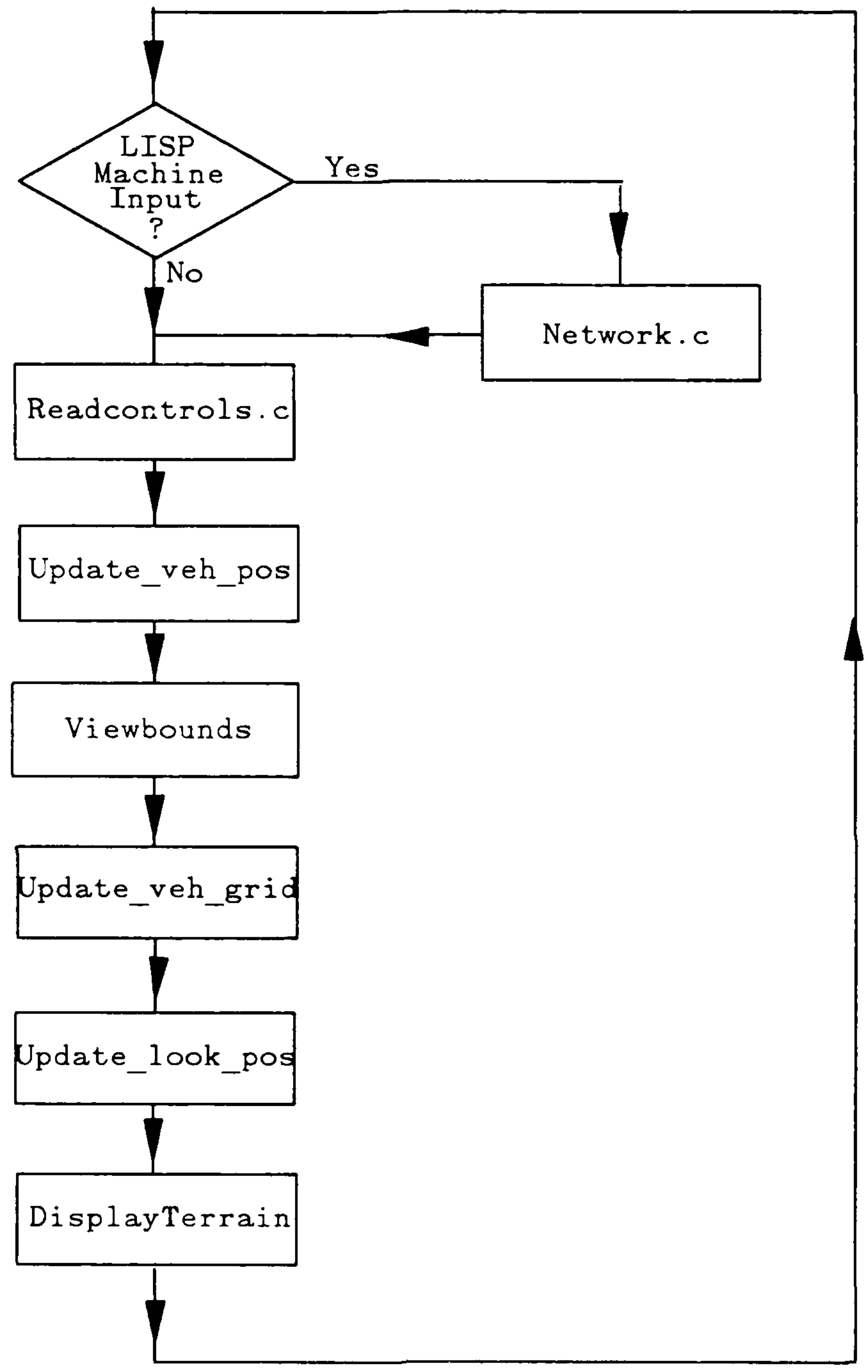


Figure 4.13 Display Loop

6. Autonomous Vehicle Control

All of the features available in the manual control mode are available to vehicle control by the LISP machine. The driven vehicle is controlled manually from the operator's console on the IRIS and each tank in the formation is controlled by its corresponding LISP Machine. The LISP Machine is capable of controlling the vehicles' courses, speeds, and viewing angles. The LISP Machine can select any vehicle and display its out-the-windshield view of the terrain and other vehicles within its field of view. The function *network.c,* shown in Appendix K, takes commands from the LISP Machine and applies them to the appropriate vehicle. The LISP Machine is continuously provided with relevant vehicle information; ie., the number and types of vehicles defined, and vehicles' courses, speeds, and positions, by the function *sendlisp.c* listed in Appendix N.

C. DRIVER AND COMMANDER SIMULATION

Driver and Commander functions are divided into two functional areas in the simulation. The software for the Driver simulation is composed of modules *tankcontrol.lisp, tankposition.lisp, tanktalk.lisp, taskgenerator.lisp, taskexecutor.lisp, and vision.lisp.* Commander functions are implemented in the Prolog module *Mett* and the KEE knowledge base *Cmd_cntrl.U.* A discussion of the functions of these modules follows.

1. Tankcontrol.lisp

Tankcontrol.lisp in Appendix A is comprised of five functions: *start-the-battle, calculate-relative-time, check-for-command, gettanks,* and *assume-control.* It is the high level module for the *Driver* or *Pilot* controller. This module must reside upon each LISP machine that controls a tank in the simulation. The *Driver* is invoked for the duration of the program run by applying *start-the-battle* to its arguments *x* and *clockvehicle.* The variable x represents the number of times the simulation performs the tank algorithm presented in Figure 3.2. *Clockvehicle* represents the baseline vehicle used to calculate the elapsed time since the last iteration and the approximate time before the next iteration of the tank algorithm. This is important because the algorithm must gauge the response time of the vehicle it controls to the task commands it produces in order to satisfy real-time goals.

Start-the-battle first observes its clockvehicle[*] and notes its position. It then initializes the algorithm response time, **next-time**[†], by applying calculate-relative-time to the argument clockvehicle. *Start-the-battle* then iterates x times. Upon each iteration, it reevaluates its response time and performs the tank algorithm. It applies check-for-command with no arguments and returns either nil, a new formation, or a unit command. If a new formation is returned, gettanks is applied to its argument *desired_formation* and new formation rules and facts are acquired from disk storage. The iteration then applies the function *assume-control.*

Assume-control invokes the major portion of the tank algorithm. It performs the visual scan of the environment by applying the function *vision* located in the module *vision.lisp* to its argument *tank,* which is the vehicle represented by this particular module, and returns *observations*. *Assume-control* then applies the function *forward-chain* from the *taskgenerator.lisp* module to its arguments *observations, tank characteristics, and *formation-rules**. *Tank characteristics* is a function application of

[*]The clockvehicle is observed to gauge the passage of time on the IRIS. The clockvehicle's speed is known and assumed constant. By observing the clockvehicle's position twice in the algorithm, the distance the clockvehicle has traveled is measured. Time is then calculated since speed and distance is known. This allows the algorithms to approximate the response times to various commands regardless of other factors such as network traffic, garbage collections, or operating system overhead.

[†]Lisp programmers often designate global variable by delineating them with astericks.

get_tank_knowledge to the argument tank. It is the facts the tank knows about itself in relation to its position in a particular formation and who its guide vehicle is.

Forward-chain produces as a side effect the global variable *tasklist* which is the list of vehicle referent velocities and directions which must be transmitted to the vehicle controller residing on the IRIS. *Assume-control* then applies function *taskexecuter* from the *taskexecuter.lisp* module to the global tasklist to communicate the referent velocities and directions to the vehicle controller. *Assume-control* then returns to *start-the-battle* for the next iteration.

Figure 4.14 presents a single iteration of *start-the-battle* for Tank 1 operating in conjunction with two other vehicles, Tank 2 and Tank 3.

2. Tankposition.lisp

This module, listed in Appendix J, contains the coordinate transformation functions applied by the function *vision* in Vision.lisp to transform object positions that are in world coordinates to referent body coordinate locations [Ref. 28] for the vehicle doing the visual scan.

3. Tanktalk.lisp

Tanktalk.lisp, in Appendix E, is the task interface layer. It is an implementation of the symbolic tasks produced by the *taskgenerator.lisp* module as function applications. It provides a logical bridge to the implementation specific TCP/IP application layers of the modules *Irisflavor.lisp* and *Symiris.lisp. Tanktalk.lisp* uses methods [Ref. 22] and [Ref. 7] created in *Irisflavor.lisp* and *Symiris.lisp* to implement the logical communication between the pilot and the vehicle controller for referent velocity and direction changes. The methods implemented in this module are part of the flavor *conversation-with-iris* which is created in Irisflavor.lisp and Symiris.lisp for their respective types of LISP Machines. These methods are sent as messages to the instance

```
> (gettanks "lineformation")
T
> (start-the-battle 1 3)

 Tank #1 now conscious

name   = 1
x1     = 5300.49172
z1     = 1743.1225
speed  = 0.0
direct = 302.25
r-angle = 57.75
name   = 2
x2     = 5408.97171
z2     = 1720.71161
speed  = 1.0
direct = 303.100006
reldir = 0.0
x2rel  = 38.9329847
z2rel  = -103.7033238
x2nxt  = 38.9329847
z2nxt  = -84.3703722
distance= 19.33295162
*next-time* == 19.33295162


(RULE CLOSE-RIGHT SAYS TASK MOVE-TO-RIGHT 1)

(TASK MOVE-TO-RIGHT 1 BECAUSE)
(RIGHT VEHICLE IS 2)
(1 IS LEFT OF 2)
(1 WILL BE LEFT OF 2)
(1 WILL BE TOO FAR FROM 2)
(GUIDE VEHICLE IS 2)
(VEHICLE IS 1)
(FORMATION IS LINE)
```

Figure 4.14 Single Iteration of Start-the-Battle

of the flavor *conversation-with-iris*. The *handle* for the instance of that flavor resides in the global variable **battle** which is instantiated during load time in the tankcontrol.lisp module. The methods implemented in this module are: *:object, :vision, :task-exec, :viewer, :lookat, :clock,* :frame-interval, and *:frame-count.*

Upon receiving the message *:object, *battle** then communicates a request to the IRIS to receive an object. Objects are lists composed of a name and a further embedded list composed of x-coordinate, y-coordinate, z-coordinate, speed in mph, and a compass direction of movement.

When receiving the message *:vision, *battle** returns an association list of objects in the tank's field of vision. Vision can be limited by constraining the objects sent by the IRIS in regard to distances or terrain constraints.

Upon receiving the message *:task-exec, *battle** relays a command to the vehicle controller for execution in the graphics simulation on the IRIS. Commands include: changing speed by a requested delta, changing direction by a requested delta, elevating and traversing the gun, or changing speed and direction by an absolute value.

The *:viewer* message changes the out of windshield view to a specified tank. A *:lookat* message requests the object list of a specific object or vehicle by name, while *:clock* and *:frame-interval* requests generate system time information from the IRIS graphics workstation, but are not used at this time.

4. Taskgenerator.lisp

The *Taskgenerator.lisp* module in Appendix C is a modified implementation of a rule-based forward-chainer developed in [Ref. 29]. The forward-chainer continually matches assertions to antecedents of a rule in the rule list, creating new assertions, which are then matched in the same manner, until all assertions and all rules have been tried. The output of this forward-chainer is the new assertions list. The forward-chainer is invoked by *assume-control* in *tankcontrol.lisp* by applying function *forward-chain to its arguments tankfacts* and *tankrules*. *Tankfacts* is a list consisting of the visual observations produced by the function *vision* in the module *vision.lisp* and the tank assertions of the individual tanks residing in the disk files for a particular type of formation. *Tankrules* consist of rules residing in the disk files for a particular type of formation. The forward-chainer's function is to produce, as a side effect, the list of tasks to execute which is assigned to the global variable *tasklist*.

Of further note are three functions in this module: *How*, *Why* and *Explain*. *How* prints the assertions that allowed the deduction of the newly created assertion by tracing the goal tree *rules-used-list* down from the newly created assertion in question. *Why* prints the assertions that depend on the assertion in question by tracing the goal tree up from the assertion in question to all the assertions that depend on it. *Explain* recursively applies the function *How* to the tasklist to produce the explanations or reasons for all of the tasks identified to be executed.

5. <u>Lineformation.lisp</u>

Lineformation.lisp in Appendix I is an example of the format of the facts and rules that are stored and retrieved for each type of tactical formation a tank can assume. This formation knowledge is retrieved by the function *gettanks* in tankcontrol.lisp whenever a new formation is dictated for the unit by the command and control module. In *Lineformation.lisp* there are six separate lists. The first five lists are individual assertions representing a specific knowledge known by each respective tank in the unit. The format for these assertion lists is provided in Figure 4.15. In *Lineformation.lisp* there are five lists that correspond to a tactical formation consisting of five tanks. Tanks can be added or deleted by adjusting the number of assertion lists in a particular formation file and adjusting the function *gettanks* in tankcontrol.lisp.

```
;;; begin list of assertions of the form
(
 ( expressions ....)
 ( expressions ....)
 ....
 ....

) ;end  list of assertions
```

Figure 4.15   List of Assertions Format

The sixth list represents the combined rule based knowledge for a tactical formation. It is shared collectively among all tanks in the simulation. These rules implement the station keeping algorithms of Figures 3.2 through 3.5. The basic format for a rule list is presented in Figure 4.16. Rules have antecedent expressions which, when correctly matched by the task generator to assertion expressions, fire precedent rule expressions. These precedent expressions are then used as assertions by the task generator. Forward chaining continues until no more assertions can be generated.

Of particular note in Figure 4.16 are the lists of the form *(> variable)* and *(< variable)* residing in the antecedent and precedent expressions of a rule. Any number of these lists can reside anywhere in an expression. They represent instructions to the function *match* which is the inference engine of the forward chaining process in *Taskgenerator.lisp*. An explanation of each symbol and its use in matching expressions is provided in Figure 4.17.

6. Taskexecutor.lisp

Taskexecutor.lisp in Appendix D contains the actual task executer and all functionally implemented tasks the individual vehicles are capable of executing. The

```
;;; begin list of rules of the form
(
 (rule <rule-name>          ;begin rule
  (if                       ;begin ifs
    ;antecedent rules
    ((> x) expressions .... (> y))
    ((< y) expressions .... (> z))
  )                         ;end ifs
  (then                     ;begin then
    ;precedent or consequents
    ((< z) expressions .... (< x))
    ((< x) expressions .... (< z))
  )                         ;end thens
 )                          ;end rule
 (rule .....)
 (rule .....)
 .
 .
 .
 )
```

Figure 4.16 List of Rules Format

tasks defined here represent the layer of abstraction between the symbolic tasks produced by the taskgenerator.lisp module and the implementation specific TCP/IP application layers of the modules *Tanktalk.lisp, Irisflavor.lisp* and *Symiris.lisp*. The task executer is invoked by applying the function *taskexecuter* to its argument *tasklist* which is the list of tasks developed by the task generator. *Taskexecuter* recursively applies *eval* to the tasklist until the tasklist is exhausted.

The tasks implemented in *taskexecutor.lisp* are organized into three functional areas; direction tasks, speed tasks, and time tasks. Direction and speed tasks allow both absolute direction changes and relative direction changes. Time tasks allow a measure of

'?' = Wild card one element in the expression. This will match any atom.

(any (? x) is a match) <= (any atom is a match)

'+' = Wild card a variable number of elements in an expression. This will match any list bounded on either or both sides of the variable with an atom.

(any (+ y) is a match) <= (any number of atoms is a match)

'>' = Remember the variable as the value of the atom found in its place.

(this (> x) is a match) <= (this atom-1 is a match)

and

'<' = Replace this variable with the value of the previous '>' for this variable and match it to an assertion.

(this (< x) i remember) <=> (this atom-1 i remember)

'>*'= Remember the variable as the value of a list of atoms found in its place. This will remember any list bounded on either or both sides of the variable with an atom.

(this (>* z) is a match) <= (this list of atoms is a match)

and

'<*'= Replace this variable with the value of the previous '>*' for this variable and match it to an assertion.

(this (<* z) i remember) <=> (this (list of atoms) i remember)

Figure 4.17 Match Instructions

system time to the overall simulation for future research. By applying the function *change-direction-to* to its arguments *tank* and *direction*, the vehicle controller module that resides on the IRIS is directed to change the direction of movement of the specified tank to the desired direction. Its velocity equivalent is *Change-speed-to* which, when applied to its arguments *tank* and *speed*, directs the vehicle controller to change the speed of the specified tank. Relative referent velocity and direction changes are implemented in functions; *Increase-speed, Decrease-speed, Move-right,* and *Move-left*. These functions are applied to two arguments representing the desired tank and the desired relative change.

The functions *Move-to-left, Move-to-right, Turn-left, Turn-right, Back-up,* Stop, and *Surge* are self explanatory. A single argument *tank* is supplied to one of these functions in order to specify to the vehicle controller for which vehicle the task must be executed.

7. Vision.lisp

Vision.lisp contains the functions that are required to simulate an individual vehicle's vision capability in the graphics environment of the IRIS. The visual scan is performed in the module *tankcontrol.lisp* by applying the function *vision* to its argument *tank* that specifies from which tank the view is requested. Vision first sends the message *:vision* to **battle** which is the handle for the instantiation of the flavor, *conversation-with-iris*. **Battle** then communicates the request for the vision scan to the vehicle controller module located on the IRIS. The vehicle controller performs the vision scan for the requested vehicle, and communicates a list of objects back to the requester. *Vision* extracts, from the list, the requester's object and then sorts the remaining list of objects by applying the function *sort-view* to the list. *Sort-view* creates a list of objects sorted by order of closest to furthest from the requesting vehicle.

The object representing the requester is then used as the basis for the coordinate system used to decide the observations. The requester's x coordinate, z coordinate, speed and direction are extracted from the object and this becomes the base for the body coordinate transformations of the other objects. The z coordinate is used instead of the traditional y coordinate because of the coordinate conventions used in the graphics simulation. For purposes of this discussion, it would be proper to consider the z axis as being equivalent to the y axis or North-South scale on a terrain map.

Once the body coordinate system is established, vision iterates over the sorted list of objects. Two vectors are calculated for each object. The first vector represents the location, speed and direction of travel relative to the base object. The second vector represents the approximate location, speed and direction of travel of the observed object at the start of the next iteration of the tank algorithm. This allows *tankcontrol.lisp* to predict and address future events. Figure 4.18 provides an example.

In Figure 4.18, *x2rel* and *z2rel* are the x and y coordinates of Tank 3 relative to Tank 1 at time *T*. The variable *x2nxt* and *z2nxt* are the predicted x and y coordinates of Tank 3 relative to Tank 1's predicted future location at time *T'*. Tank 1 will be too close to Tank 3 because the horizontal interval distance will exceed the value of **proper-interval** as Tank 1 approaches Tank 3 from behind. A task is then generated by *tankcontrol.lisp* to avoid the undesirable state.

Once an object's vectors are calculated, *vision* goes about it's main task, asserting facts about the environment. Facts are asserted each time the established criteria for a test is met. Tests are organized into areas of interest based upon the requirements of a particular research topic. The version of *vision.lisp* in Appendix B is tailored specifically for the requirements of station keeping. Other versions contain tests

```
Tank #1 now conscious

name   = 1
x1     = 4474.45999
z1     = 2411.60965
speed  = 1.468625
direct = 302.25
r-angle = 57.75
name   = 3
x2     = 4453.29446
z2     = 2428.31863
speed  = 1.0
direct = 302.25
reldir = 0.0
x2rel  = 2.83701507
z2rel  = 26.81643313
x2nxt  = 2.83701507
z2nxt  = 7.177394718
distance= 19.63903843
*next-time* == 41.90779063

(RULE AVOID-COLLISION-TO-RIGHT SAYS TASK MOVE-TO-LEFT 1)
(RULE DIRECTIONS SAYS LEFT IS OPPOSITE OF RIGHT)
(RULE DIRECTIONS SAYS RIGHT IS OPPOSITE OF LEFT)

(TASK MOVE-TO-LEFT 1 BECAUSE)
(1 WILL BE LEFT OF 3)
(1 WILL BE TOO CLOSE TO 3)
(VEHICLE IS 1)
(FORMATION IS LINE)
```

Figure 4.18 Reasoning about Future Events

and criteria for topics such as terrain appreciation, threat identification, and battle simulations.

The output returned from vision is a list of facts. Some possible facts that can be asserted by vision are presented as the top 25 functions in Appendix B. Typical functions include: toofar, tooclose, rightof, leftof, forward, behind, aheadof, online, oncourse, and faster. When applied to their arguments x and y, they return a fact to be cons'd to the list **observations**, which is returned to the function *forward-chain in taskgenerator.lisp* when it applies vision.

8. <u>Command and Control Inferencing</u>

The command and control module is a Prolog program that is executed on the Vax 11/785. It utilizes the forward chaining programs developed in Prolog [Ref. 30] to acquire and assert information required to deduce three critical elements of a tactical plan. These elements are: the formation the unit must assume, the targets to be addressed by supporting fires, and the method of attacking an objective. By querying the rule *go* with the variables *Formation, Fireplan,* and *Attackplan,* the tactical assessment functions are invoked. The program then queries the user for information concerning the mission, enemy disposition, terrain and weather, and what type of supporting assets the unit has. User input is acquired as a list of menu items that were selected. Using this list, facts are asserted and forward chaining occurs. The user is prompted for information until the command and control module has enough information to deduce a plan. The output is in standard Prolog format for variable unifications.

The command and control module can determine any of six types of standard formations, targets of opportunity or upon the objective, and either a frontal assault or single envelopment for a plan of attack. Typical considerations include the phase of movement, terrain features in the axis of advance, and types of weapons for both friendly and enemy forces. The command and control module is presented in Appendix L.

D. HARDWARE AND SOFTWARE INTERFACES

Hardware and software interfacing is accomplished by the TCP/IP and CHAOS application layers of Appendices F through G. The TCP/IP applications layers reside in the modules *Irisflavor.lisp* and *Symiris.lisp*. These modules create the flavor *conversation-with-iris* for the TI Explorer and Symbolics Lisp Machines respectively. They are designed to implement the Inter-Computer Communication Package protocol described in [Ref. 31].

There are four standard methods for the TCP/IP applications layers: *:start-iris*, *:get-iris*, *:put-iris*, and *:stop-iris*. The method *:start-iris* establishes a client session to the IRIS server host. The methods *:get-iris* and *:put-iris* receive and send messages respectively. The messages can be of type integer, float, character or string. The message *:stop-iris* terminates the connection with the server host. The complimentary functions that must reside on the host IRIS are found in the vehicle controller software of Appendices K and N. See [Ref. 31] for detailed instructions and use of the Inter-Computer Communications Package.

The CHAOS application layer of Appendix G has a similar set of methods in its flavor, *my-chaos*. The methods are *:start-user*, *:start-server*, *:get*, *:put*, and *:stop*. The methods *:start-user* and *:start-server* require a host name of type host object and a contact name which is of type string. The host name must be known to the Chaos network of the communicating machines and represents the target host. The contact name can be any unique string of characters and represents the identifier for a session. The CHAOS application layer implements Layer 7 of the Open System Interface standard as does the two TCP/IP applications layers. However, Layers 4-6 are different. Chaos Layers 4-6 are faster than TCP/IP although slightly less reliable.

E. USERS GUIDE

1. Initial Startup

The IRIS portion of the simulator is menu driven, with help information displayed with each of the on-screen menus. The IRIS system is initialized prior to initializing the LISP Machines. The IRIS part of the simulator is started by typing in "veh" from the appropriate directory. Then the LISP Machines are started by typing in (load "thesis") on each of them. This loads in the LISP modules. Then the user is prompted for the desired formation. The choices are "C" for column or "L" for line formations. The user is then prompted "start networking?". The user then enters "Y". The routine *thesis.lisp* sets the tank algorithm to iterate 100 times.

Entering "veh" on the IRIS loads the simulator and preprocesses the terrain data. The terrain is read from a file named *dted.veh.* This file is stored in directory DTED. If the file cannot be found, the simulator displays a warning and asks if the user wants to continue with flat terrain or to quit. Without terrain data, the terrain is drawn at zero elevation giving a flat, checkerboard appearance. The file *polygon.data* is then read from directory DTED. This is the file of terrain polygon colors and is created if not found in directory DTED.

At this point the opening menu and the first of the three introductory screens appear. Menu selections are made by positioning the cursor, with the mouse, over the desired menu selection, and then pressing the left mouse button. The six main menus available to the users for simulator control are: Opening Menu, Main Menu, Add Vehicle Menu, Delete Vehicle Menu, Switch Vehicles Menu, and the Run Menu.

a. Opening Menu

The menu choices provided by the opening menu are: Next Page, Previous Page, and Quit Program. The introductory screens can be paged through by selecting the next page or previous page options. The user can quit the program at this time by selecting the quit program option.

b. Main Menu

The options provided by the opening menu are: Add Vehicle, Delete Vehicle, Defaults, Run, Zoom in/Out, and Quit Program. After paging through the introductory screens, the main menu appears along with a two-dimensional contour map of the terrain (Figure 4.6). All the vehicles used in the simulator are defined and initialized from the main menu.

Vehicles are added by selecting the *add vehicles* option or the *defaults* option. Add vehicles allows the operator to select the vehicles (tanks, jeeps, trucks), their locations on the map, and initial speeds. A vehicle's location is set by moving the cursor to the desired location on the display map and pressing the left mouse button. The cursor then gives a rubber band line from the vehicle icon on the map to the current cursor position. This line represents a possible course which is set by pressing the left mouse button. Once the course has been set, a slider speedometer appears (Figure 4.8). The speed is set by sliding the rectangle on the speedometer to the desired speed and pressing the left mouse button. The default option places one of each type of vehicle (tank, jeep, truck) near the middle right area of the terrain, all on a course of zero degrees and with a speed of twelve miles-per-hour.

The operator can delete a vehicle by selecting the delete vehicle option, placing the cursor over the vehicle to be deleted and pressing the left mouse button. Zoom in/out allows the operator to view a small area of the contour map. When the operator is finished with vehicle definition and program initialization, selecting the run option starts the simulation. The operator moves the cursor over the vehicle that is selected to be the driven vehicle and presses the left mouse button. This causes the simulator to enter the display loop.

2. Vehicle Controls

Once the display loop is entered, the display changes to the view from the inside of the driven vehicle. The driven vehicle can be controlled by the mouse, the dial box, or from control information transmitted by the LISP Machines. Courses, speed, steering angle, viewing angle, and tilt can be manually controlled by the operator from the dial box. The viewing angle can only be changed when the steering angle is zero. The apparent viewing volume can be changed by holding down the middle or right mouse button. The driven vehicle's position on the terrain is always shown on the contour map displayed in the middle, right portion of the display screen (Figure 4.1). Digital readouts of the driven vehicle's ordered speed, actual speed, course, view angle, steering angle and degree of zoom are always visible on the lower, right part of the display screen (Figure 4.1).

The LISP Machines can control the motion of the tank vehicles over the terrain. The commands currently implemented are relative and absolute course changes, relative and absolute speed changes, and changing the display perspective from one vehicle to another.

F. SUMMARY

The software and communication interfaces for the three-dimensional graphics simulator and autonomous vehicle controllers are discussed in this chapter. The driver and commander simulation functions are decomposed and explained in detail. A users guide is included to facilitate operation of the simulation.

V. EVALUATION

A. INTRODUCTION

The fundamental requirements of the prototype Simulation System for Combat Vehicle Coordination and Motion Visualization (SSCVCMV) are to allow rule system modeling of command and control aspects of small unit behavior using current doctrine and to provide real-time graphics motion visualization of the model.

Collateral research in four areas of specific interest is being pursued. First, is it possible to model the complex motion interaction of a small tactical unit of combat vehicles during the planning phase, movement to contact, deployment, execution, and consolidation phase of a typical mission? Second, can the model sufficiently assume the characteristics of a tactical unit operating autonomously in a threat environment? Third, are the Iris/Symbolics machines and current interface technology capable of simulating the real time environment with their existing architecture? Last, is the model sufficiently close to current tactical mobility behavior to warrant further development by DoD?

Individual tanks of the Simulation System for Combat Vehicle Coordination and Motion Visualization (SSCVCMV) perform functionally according to the algorithms presented in Chapter 3. The pilot possesses additional capabilities, besides those being developed at FMC [Ref. 8], to allow the vehicle to act as an integral part of a tactical autonomous unit. Based upon commands sent to the vehicle by the lead vehicle, it assumes its designated place in a tactical formation and keeps its station with the formation until further commanded. Currently, the tanks use three sets of simple rules that allow the vehicles to assume a line, column, or file formation. For each formation, each tank possesses knowledge about who it is, the type of formation, its guide vehicle, and the vehicles that should be to its flanks, front and rear.

An autonomous tank is comprised of a set of functions that reside upon a LISP machine, its vehicle controller which resides upon the IRIS, and its graphic tank object which also resides upon the IRIS. Each LISP machine controls a graphically rendered tank on the IRIS battlefield during a simulation run. These Lisp functions perform the algorithms presented in Chapter 3 using rule system modeling. They also gauge the response time of the vehicles they control on the IRIS to the task commands they produce in order to satisfy real-time goals in the IRIS battlefield environment.

The tanks perform a simulated visual scan of the environment in the IRIS and produce high-level observations about the battlefield. These observations are used to perform tactical assessments and create tasks to accomplish goals using rule-based inferencing engines.

Typical tasks, such as those generated for formation keeping goals, are vehicle referent velocities and directions. These tasks are transmitted to the vehicle controller residing on the IRIS. The vehicle controller then executes the tasks and communicates feedback information to the requesting Lisp machine.

The tanks reason about the IRIS battlefield world relative to their own individual body coordinate systems. The tanks also reason about time by approximating positions, dispositions, and possible intentions of objects in view during possible future event time frames. Tanks also continuously re-evaluate their individual circumstances as well as their vehicle controller's response time to a direction or velocity command. This allows a tank to predict and address future events. Figure 5.1 provides an example.

In Figure 5.1, *x2rel* and *z2rel* are the x and y coordinates of Tank 3 relative to Tank 1 at time *T*. The variable *x2nxt* and *z2nxt* are the predicted x and y coordinates of Tank 3

```
Tank #1 now conscious

name   = 1
x1     = 4474.45999
z1     = 2411.60965
speed  = 1.468625
direct = 302.25
r-angle = 57.75
name   = 3
x2     = 4453.29446
z2     = 2428.31863
speed  = 1.0
direct = 302.25
reldir = 0.0
x2rel  = 2.83701507
z2rel  = 26.81643313
x2nxt  = 2.83701507
z2nxt  = 7.177394718
distance= 19.63903843
*next-time* == 41.90779063

(RULE AVOID-COLLISION-TO-RIGHT SAYS TASK MOVE-TO-LEFT 1)
(RULE DIRECTIONS SAYS LEFT IS OPPOSITE OF RIGHT)
(RULE DIRECTIONS SAYS RIGHT IS OPPOSITE OF LEFT)

(TASK MOVE-TO-LEFT 1 BECAUSE)
(1 WILL BE LEFT OF 3)
(1 WILL BE TOO CLOSE TO 3)
(VEHICLE IS 1)
(FORMATION IS LINE)
```

Figure 5.1 Reasoning about Future Events

relative to Tank 1's predicted future location at time *T'*. Tank 1 will be too close to Tank 3 because the horizontal interval distance will exceed the value of a constant measure called *proper-interval* as Tank 1 approaches Tank 3 from behind. A task is then generated by Tank 1 to avoid the undesirable state.

The system is distributed across the various specialized architectures in accordance with hardware capabilities. Thus, it was possible to create an extremely satisfactory real-time system at low cost. The current suite of equipment allows up to five individual tanks to operate on the battlefield of the IRIS.

B. A TYPICAL TEST MISSION FOR THE SSCVCMV

1. IRIS Initialization

A typical test mission for the SSCVCMV consists of initializing the *battlefield* as to placement of the unit of vehicles, the type of vehicles of the unit, and their starting disposition. Figure 4.6 shows the *battlefield* in an overhead display mode prior to adding vehicles using a mouse driven menu selection. Figures 4.7 and 4.8 show a vehicle being added to the test mission.

In Figure 4.7, the vehicle (in this case a tank) is given an initial position and direction. In Figure 4.8, the same vehicle is then subsequently given an initial speed.

Figure 5.2 represents side terminal output of the IRIS noting that upon initialization with the command 'veh', internal addresses are allotted for communication connections to the LISP machines. The addition of tanks to the battlefield is also recorded. The required number of vehicles are added to the battlefield iteratively in this manner. There is no programmed limit to the number of vehicles or LISP machine connections in the system. However, using the IRIS 3120, a real-time limit of five LISP machine connections is imposed because of network load.

Figures 5.3 and 5.4 represent fairly typical tank dispositions at the start of a session. The graphics simulation screen is divided into two main areas, the out-of-the-windshield view, and the information panel. A dialbox control and mouse is not pictured, but are integral parts of the control function of the system. Operating menu items can be executed at anytime during the simulation by manipulating the mouse. The

```
Script started on Thu Apr 21 10:46:12 1988
IRIS2 1% veh
lisp is at addr 101dc0 in main
before the open to machine
startshared= 1937408
segment address= 1d9000
startshared= 1937932
segment address= 1d920c
startshared= 1938456
segment address= 1d9418
startshared= 1938980
segment address= 1d9624
after the open to machine
tank->name = 1              /* tank 1 is added */
tank->name = 2              /* tank 2 is added */
tank->name = 3              /* tank 3 is added */
tank->name = 4              /* tank 4 is added */
tank->name = 5              /* tank 5 is added */
tank->name = 6              /* tank 6 is added */
tank->name = 7              /* tank 7 is added */
IRIS2 2% exit
IRIS2 3%
```

Figure 5.2 Side Terminal Output of IRIS

reduced map in the information panel records the direction and position of the viewing vehicle by the red arrow on the map. The red vectors to either side of the arrow indicates the view "window" of the viewing vehicle. The two green boxes in the lower right corner represent control device options while the speed and course information is recorded for the current viewing vehicle.

Figure 5.3 shows a large force of tanks in a column formation . A test mission might be started in this phase, which occurs during approach march to, or departing from, a tactical control point called the *assembly area*. The view is from over the hood of a stationary jeep. The speed of the jeep is currently 0, the jeep is on a course of 107 degrees, while the center of the viewing angle is at 107 degrees.

Tanks are not constrained to starting in column formation. Figure 5.4 is the view from a jeep showing a stationary force of tanks, online, across the forward slope of a hill already stationed at the *assembly area*.

Once the IRIS has been initialized for the current test scenario, the tactical assessment of the situation is conducted.

2. Tactical Assessment of the Situation

Currently, the tactical assessment stage is represented by an interactive session on the Vax 11/785. Figure 5.5 shows a session conducted for a typical test mission.

The required output from this segment of the command and control phase is the formation for the unit to assume, the preplanned supporting fires, and the method of assault. The formation is the order of movement from the point of crossing the line of departure at the assembly area, through the movement to contact phase, until the unit reaches the final coordination line. Preplanned supporting fires are those fires that are planned upon key terrain and expected enemy locations to mask the movement of the maneuvering unit. The method of assault (in this case frontal) is the method of maneuver against an objective during the movement to contact.

This implies that the unit will proceed from the assembly area, cross the line of departure, and conduct the movement to contact in a column formation. This is the optimal formation when the enemy threat and direction is unknown, speed is desired,

The required output from this segment of the command and control phase is the formation for the unit to assume, the preplanned supporting fires, and the method of assault. The formation is the order of movement from the point of crossing the line of departure at the assembly area, through the movement to contact phase, until the unit

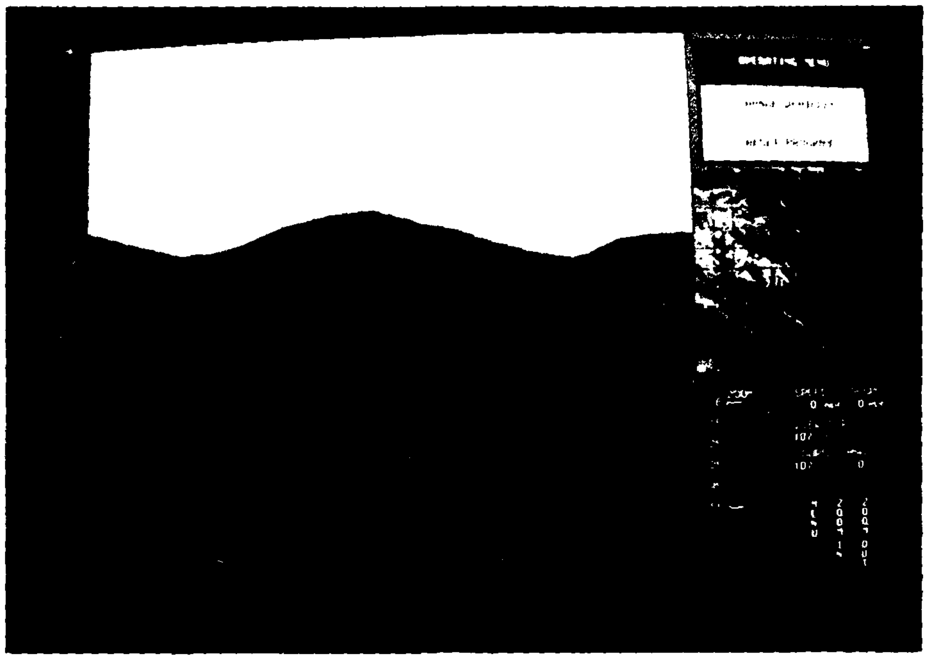


Figure 5.3 Tanks in the Assembly Area

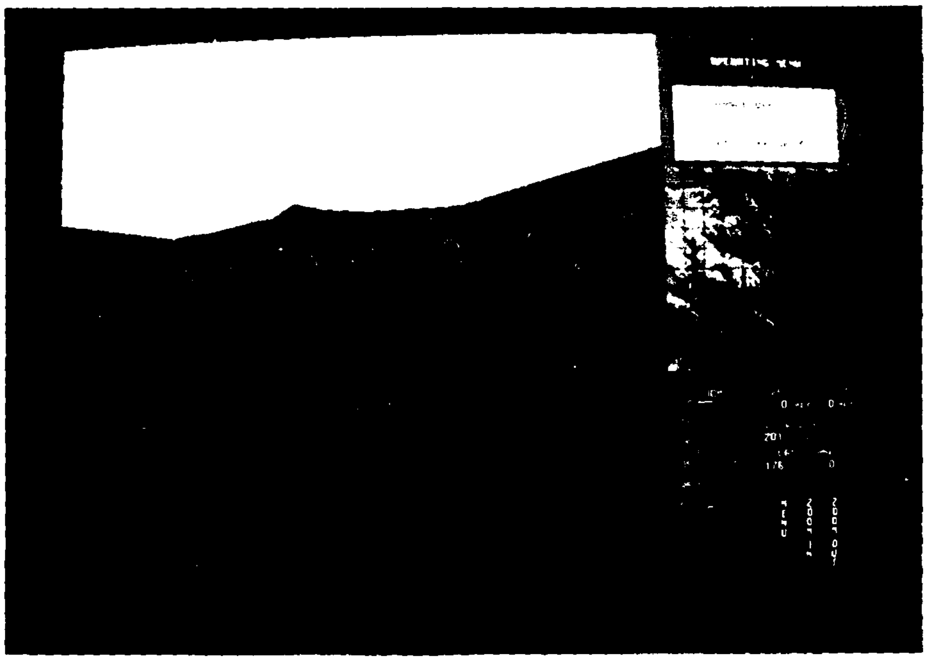


Figure 5.4 Tanks On Line

```
Script started on Wed Apr 20 12:17:32 1988
UNIX1 1% prolog

C-Prolog version 1.5+
| ?- [mett].
menu consulted 3188 bytes 0.65 sec.
forward consulted 3380 bytes 0.700001 sec.
utility consulted 1744 bytes 0.350001 sec.
nmett consulted 14320 bytes 3.25 sec.

yes
| ?- go(Formation_to_assume,Call_for_fire_on,Method_of_Assault).

1: The unit is moving to_contact?
2: The unit is moving to_assembly_area?
3: The unit is moving cross_open_area?
4: The unit is moving to_cross_line_of_departure?
5: The unit is moving to_final_coordination_line?
6: The unit is moving to_objective?
7: The unit is attacking an objective?
Give numbers of questions whose answer is yes.[4,5,6,7].

  .
  .
  .

Formation_to_assume = column
Call_for_fire_on = objective
Method_of_Assault = frontal

yes
| ?- halt.

[ Prolog execution halted ]
UNIX1 2% exit
script done on Wed Apr 20 12:24:55 1988
```

Figure 5.5 Tactical Assessment of the Situation

reaches the final coordination line. Preplanned supporting fires are those fires that are planned upon key terrain and expected enemy locations to mask the movement of the maneuvering unit. The method of assault (in this case frontal) is the method of maneuver against an objective during the movement to contact.

This implies that the unit will proceed from the assembly area, cross the line of departure, and conduct the movement to contact in a column formation. This is the optimal formation when the enemy threat and direction is unknown, speed is desired, and the opposing force is an inferior force. The unit will proceed to the final coordination line under the masking fire of its supporting artillery until it reaches the final coordination line. Once reaching the final coordination line the unit will deploy on line and assault through the objective. This completes the current implementation of the tactical assessment phase. The final product of this phase was the determination of the formation, the preplanned supporting fires, and the method of attack.

3. LISP Machine Initialization

Once the tactical assessment phase has been conducted the LISP machines that provide the artificial intelligence capabilities for the tanks in the unit are initialized. Figure 5.6 is the screen of a Lisp machine being initialized for the test mission. The currently executable formations (those which rules are written for) are column, line, and file. The prompt in the figure asks to install the default. If *no* is input, a line formation is then assumed. Once networking is signaled to begin, the LISP machine assumes control of its specified tank.

It is possible, and sometimes desirable, to create more graphics instantiations of vehicles than there are LISP machines to guide them. In this case the vehicles instantiated maintain the constant speed and direction with which they were initialized.

When approaching a slope of more than 13 degrees, they automatically initiate a slow turn to the left.

Alternately, it is quite possible to have two LISP machines involved in guiding one vehicle. All communications and operations between the LISP machines and the IRIS are atomic. Different communications processes must be established, but two LISP machines can send and receive commands for the same named tank. Thus, it could be desirable to initiate the control algorithm for the same tank on different LISP machines, staggering the execution so that the logical processes of the algorithms are running concurrently. Alternately, the processes of the algorithm could be divided into orthogonal functions and executed in a distributed manner on different machines.

```
> (login 'nelson 'lm t)
T
> (load "thesis2")
; Loading aries: NELSON; THESIS2.LISP#> into package USER
; Loading aries: NELSON; VISION.XFASL#> into package USER
; Loading aries: NELSON; TANKPOSITION.XFASL#> into package USER
; Loading aries: NELSON; TASKGENERATOR.XFASL#> into package USER
; Loading aries: NELSON; TASKEXECUTOR.XFASL#> into package USER
; Loading aries: NELSON; IRISFLAVOR.LISP#> into package USER
; Loading aries: NELSON; TANKTALK.XFASL#> into package USER
; Loading aries: NELSON; TANKCONTROL.XFASL#> into package USER

columnformation ? yes.
start networking ? yes.

> (dribble)
```

Figure 5.6 Test Mission Initialization

It is also quite possible to have one LISP machine guide an unlimited number of tanks, although real-time considerations limit this capability to two or three per Symbolics LISP machine and no more than one for the TI/Explorer. Refer to Appendix A and the *Start-the-Battle* function to see how this is done. In *Start-the-Battle* the function *Assumecontrol* is applied to its argument *tank* in order to execute one iteration of vision, task generation, and task execution for a tank. *Start-the-Battle* in Appendix A has four of these function applications commented out. By uncommenting these function applications , *Start-the-Battle* can control tanks 1 through 5.

Various scenarios have been tested, using different combinations of the above methods to either increase the number of guided vehicles in the simulation or increase the speed of response according to different criteria in the test. But a detailed discussion of this issue is not within the scope of the current study.

4. The Conduct of The Test Mission

Currently, the functions of path planning and tactical control are conducted by the human operator controlling the vehicle in the unit designated as the unit leader. Tactical control consists of determining halts for the unit or signaling formation changes at different control points. The human operator can select any vehicle to view or drive. Vehicles that the operator has selected, and which are driven simultaneously by LISP machines, obey control commands from both sources. This can become quite confusing if the commands are contradictory. The human operator uses a combination of the dialbox control system and the mouse to operate his vehicle.

Figures 5.7 through 5.10 illustrate a typical test mission. Figure 5.7 depicts the movement from an assembly area. The initialization phase for the IRIS has been conducted, the tactical assessment carried out with the results as discussed above, and four LISP machines have been initialized to drive four of the tanks in the unit. The guide vehicle for the unit, (driven by a human operator) has been given an initial direction and speed. The jeep was then selected to view the formation as it turned to its left to assume a column formation. The picture was taken from the jeep.

Figure 5.8 depicts the column after crossing the line of departure and conducting movement to contact. The guide vehicle is the lead tank in the column. To obtain the picture, the jeep was driven to a known destination of the lead tank. The jeep then was positioned to get a view as the column approached.

Figure 5.7 Moving to the Line of Departure

Figure 5.8 Crossing the Line of Departure

Figure 5.9 depicts the actions at the Final Coordination Line. The unit deployed into a line formation and is about to move through the objective. This deployment was effected with the help of manual intervention. The guide tank was stopped at the Final Coordination Line by a human operator. This forced the column to halt by firing certain station keeping rules. The function application of *Start-the-battle* was allowed to expire upon each LISP machine. A new formation was then acquired by each LISP machine by applying the function *Gettanks* to its argument *"Lineformation"*. The function *Start-the-battle* was then re-applied upon each LISP machine. The human operator assumed control of the guide vehicle while the autonomous, LISP machine

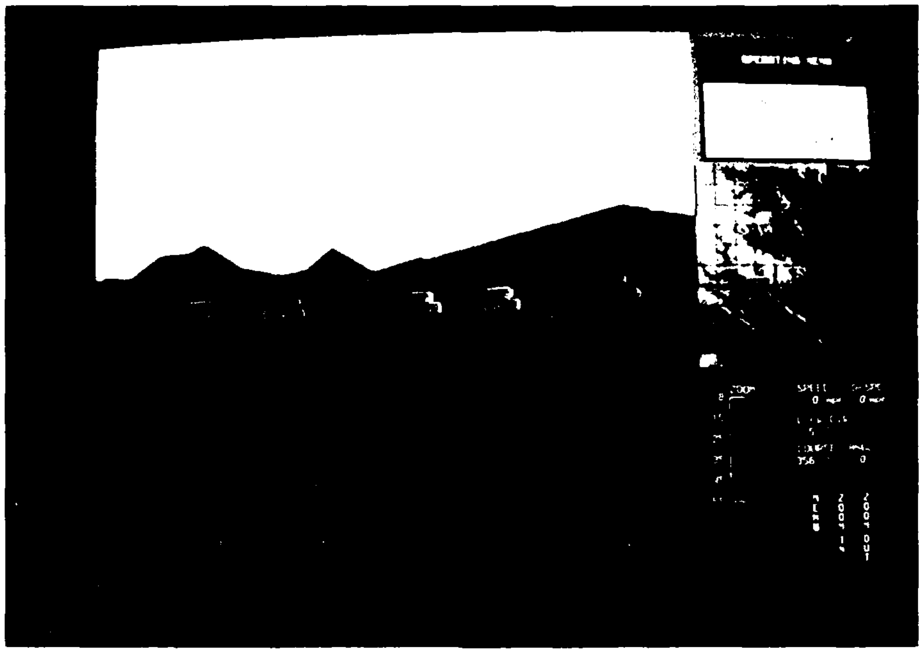


Figure 5.9 Deploying at the Final Coordination Line

driven tanks then assumed their positions in the line formation after about 30 seconds of maneuvering.

Figure 5.10 depicts a flanking view of the line of tanks as they assault an objective. The line is sweeping past the stationary jeep from which the picture was taken. The fifth tank in the line decided to maneuver around the other side of the jeep and is not depicted.

C. CRITIQUE

The prototype system is able to operate with a high degree of realism. With that said, there are still a few design decisions that need to be reexamined.

The hidden surface elimination techniques used in this simulation have a few drawbacks. One, when vehicles move along 100 meter grid boundaries the image of the vehicle can flash. This is caused by the vehicle being painted over by the terrain when the vehicle moves back and forth along the grid square edge without crossing the threashold. The threashold determines when to draw a vehicle in a new grid square.

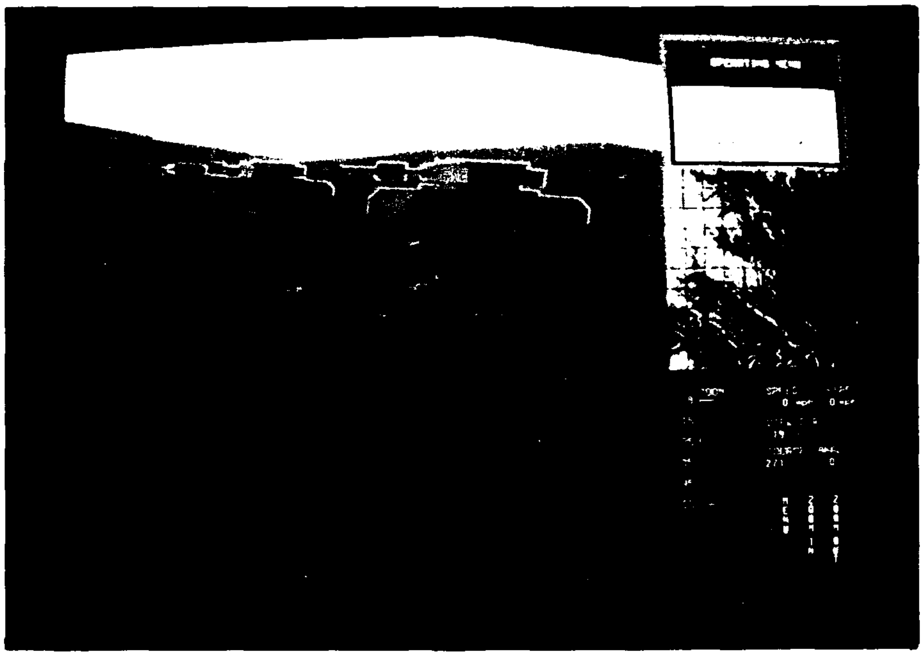


Figure 5.10 Assaulting the Objective

Two, when the view angle exceeds 17 degrees vertical from a horizontally stablized platform, the out of the windshield view distorts so as to render the picture unidentifiable. Three, when zooming in, some terrain will be drawn in yellow vice the default green. The drawbacks will be eliminated when the program is ported over to the Silicon Graphics, Inc. IRIS 4D. The IRIS 4D allows double buffered Z-buffering for hidden surface elimination.

Because of the sequential nature of the algorithm that implements the tank behavior and the slower processor of the TI Explorer, the performance of the tank guided by the TI Explorer has a noticeable lag in capability compared to the other LISP machines. In fact, it invariably is the culprit in any type of accidental collision involving tanks in a formation. To be fair, it should be noted that the other LISP machines are operating at processor speeds of almost four (4) times the TI Explorer's capability and that the Explorer II upgrade could conceivably solve the problem.

The simple dynamics of the manually controlled vehicle provide a sensitivity to control and corresponding reaction time that is probably too 'ideal'. The realism of the bounce (what is seen in the picture) does not correspond to the difficulty that should be encountered (but isn't) when attempting to control the vehicle over rough terrain.

The communications packages of the prototype system are too complex, require immense detailed knowledge that detracts from the research goals, and are liable to spurious behavior from the generally shared network of the current laboratory. The packages need to be further abstracted to a convergent level of commonality that is easily learned and easily used. The system should use a closed and dedicated communications network.

The limiting factor in the existing research is the sequential nature of the implementations of the algorithms on the typical Von Neumann architectures of the existing hardware. This limiting factor is readily apparent in any test run of the system and is the main cause of tank behavior errors within the prototype.

D. SUMMARY

The prototype system has successfully performed all initial requirements. The command and control modules can successfully decide upon a tactical formation for the unit to assume and a simple attack plan to follow. Communication interfaces have been developed to allow coordinating flows of information from the command and control modules to individual tanks of the unit. Complex motion interactions of small tactical units during any phase of a mission is graphically represented in the form of dynamically updated out of the windshield views from any vehicle operating on the terrain battlefield.

VI. SUMMARY AND CONCLUSIONS

A. RESEARCH CONTRIBUTIONS

This thesis has presented a prototype Simulation System for Combat Vehicle Coordination and Motion Visualization. This thesis developed a computerized testbed as a contribution to an ultimate goal of allowing autonomous vehicle systems to operate as organic tactical units in threat environments. The main areas of concentration in the model have been to create a platform for implementing and testing rule system modeling of command and control aspects of small unit behavior (using current doctrine) and to provide real-time graphics motion visualization of the model.

The prototype has successfully demonstrated the capability to model complex motion interaction of a small tactical unit of combat vehicles during the various phases of a typical mission using rule system modeling for command and control. The model has also successfully demonstrated the capability to assume the characteristics of a tactical operating unit autonomously by establishing and maintaining a currently practised tactical formation. It has also demonstrated that the IRIS/Symbolics machines and current interface technology are capable of simulating a more than adequate real time battle environment using low cost graphics hardware. Finally, it is proposed by the authors that this model is sufficiently close to satisfying a recognized requirement for tactical mobility behavior in both the GATORS and ALV programs as to warrant further development by DoD.

The Autonomous Vehicle Motion Simulator is an important visualization tool for rule system modeling of command and control aspects of small unit behavior. It is an inexpensive, interactive, real-time simulator that can be used as a test platform for mobility expert system algorithms.

The development of expert system based coordination algorithm for tactical units of autonomous vehicles is an important contribution of this study. This study demonstrates the feasibility of using four different computer architectures networked together to form one integrated mobility expert system.

Another important product of this work is the development of realistic vehicle dynamics incorporated into a three-dimensional, graphics simulation. The three-dimensional graphics simulation was networked to LISP Machines that served as vehicle autopilots.

This model provides a realistic environment that can be used to support real-time experimental research. The model was written in modular form and can be easily modified, enlarged, or enhanced to facilitate future work.

B. SUGGESTIONS FOR ADDITIONAL RESEARCH

1. Battle Management Scenarios

There are a number of extensions that can enhance the capabilities of the autonomous vehicle simulator. Task conflict resolution would identify and resolve illogical task sequences. This would allow the simulator to perform more sophisticated missions. Pathfinding could be incorporated to provide a mechanism for determining optimal paths to the objective based on various constraints such as: location of the enemy, fuel economy, and time. A higher level command and control processing algorithm could be developed to conduct *battlefield* like maneuvers, including, attacking and defensive scenarios. Threat and target identification procedures and an extension of the vision module would allow the simulator to accommodate more realistic combat simulations.

2. Natural Language Understanding

The command and control module should develop a simple natural language understanding capability. The goal of this capability is to communicate to the command and control module the way a tactical commander would communicate to a subordinate leader - using the standard five paragraph operations order [Ref. 13].

This is not as hard as it would first appear. Military commands, even in detailed operational orders, are laconic. In tactical situations, most ideas and actions can be communicated using subsets of a vocabulary comprised of, at most, 350 to 500 high frequency words. Operational orders are frequently scripted by subordinate leaders in preparation to receive an order. The blanks are then filled in and the order processed to be given to more subordinate levels in the chain. This suggests two alternative and complimentary ways to pursue natural language understanding in the system.

One, develop grammers using the 350 to 500 high frequency words for input to an Augmented Transition Network parser [Ref. 32]. The semantic knowledge representations would be stored in slot notation form, and the cycle of planning for subordinate leaders (in this case the command and control module) produced by a rule system. An Augmented Transition Network parser and an example grammer appear in Appendix R. The slot structure for a command module and five paragraph order appear in Appendix S.

Secondly, an Augmented Transition Tree compiler could be developed based on the known scripts for a standard five paragraph order [Ref. 29]. The compiler would construct known functions to be applied based upon the contents of the script. The command and control module would then apply those functions in the prescribed manner in order to effect its goals.

3. Introduce Hardware Concurrency to the System

Performance bottlenecks occur during communication processing on the IRIS. This is because each tank spawns a send and receive process to communicate to a Lisp Machine. However, the new IRIS 4D is also four times faster than the current battlefield, which means the system will now operate with twenty tanks.

The performance bottlenecks on the Lisp machine side are in relation to the sequential nature of the algorithm execution. The problem is that the vision and inferencing is not concurrent nor continuous. The solution is to specialize and distribute those functions across a larger suite of hardware. A possible (and expensive) approach would be to investigate the use of a CONNECTION Machine running LISP* which is a concurrent Lisp.

4. Introduce Multi-Level Conceptual Concurrency to the System

The specialization and distribution of functions suggests a *blackboard* model or approach. The tank algorithm would be implemented with separate processes for vision, command and control communication, task generation, and task execution. Interprocess communication composed of message passing would allow the various interchanges of information required between the *blackboard* and the concurrent processes. A task resolver would arbitrate any conflicting tasks sent from the blackboard to the vehicle controller. Similarly, the command and control algorithm would also benefit in the same manner by this approach. A tactical arbiter would function additionally to ascertain and address immediate problems through a type of interrupt mechanism.

5. Improve Terrain

There are a number of enhancements that could be used to provide more realistic terrain and environmental conditions. The addition of rocks, trees, and shrubs would provide cover and concealment for vehicle movement. Rivers, ravines, and other obstacles would test obstacle avoidance and path planning algorithms.

6. Realistic Lighting Model

The lighting model is constrained by the current hardware. A more realistic lighting model can be developed since the program has been ported to the IRIS 4D workstation.

7. Improved User Interface

A viable alternative to the natural language understanding capability is to create a series of menu driven interfaces using KEE's graphics capability. The user would begin by choosing the type of mission desired. This would generate further sub menus which would generate further menus until all necessary information to kick off a mission is extracted from the user. The drawback to this approach is that it requires the field commander in the ultimately developed production model to have to learn a set of unfamiliar skills to deal with this new type of subordinate unit commander. It would be better to pay the research costs for the natural language capability now in the early phases rather than try to retrofit it in reaction to field commanders complaints of needless complexity to operate it.

8. Improved Vehicle Model

The use of a full three dimensional, six degree of freedom vehicle model calculating real bounce would establish a high degree of accuracy in vehicle response for the system. For a discussion of the formulas required for the calculations see [Ref. 27].

LIST OF REFERENCES

1. Nitao, J. J. and Parodi, A. M., "A Real-Time Reflexive Pilot for an Autonomous Land Vehicle," *IEEE Control Systems Magazine*, v. 6 , no. 1, Feb 1986.
2. Tan, C. H., *A Simulation Study of an Autonomous Steering System for On-Road Operation of Automotive Vehicles,* M. S. Thesis, pp. 31-45, Naval Postgraduate School, Monterey, California, December 1986.
3. Dolezal, M. J., *A Simulation Study of a Speed Control System for Autonomous On-Road Operation of Automotive Vehicles,* M. S. Thesis, pp. 82-89, Naval Postgraduate School, Monterey, California, June 1987.
4. Kotas, J. and Reynolds, C. W., "Flocks, Herds, and Schools: A Distributed Behavioral Model," *Computer Graphics*, v. 21 , no. 4, Jul 1987.
5. Smith, D. B. and Streyle, D. G., *An Inexpensive Real-Time Interactive Three-Dimensional Flight Simulation System,* M. S. Thesis, p. 36, Naval Postgraduate School, Monterey, California, June 1986.
6. Oliver, M. R. and Stahl, D. J. Jr, *Interactive, Networked, Moving Platform Simulators,* M. S. Thesis, pp. 12-20,81,85,87,95, Naval Postgraduate School, Monterey, California, December 1987.
7. Parodi, A. M. and Moon, D. A., "Object-Oriented Programming with Flavors," *Proceedings of the First Annual Conference on Object-Oriented Programming System, Languages, and Applications, ACM*, v. 1, St. Louis, Missouri, 1986.
8. Applications of Artificial Intelligence, Mitchell, J. S. B., "An Autonomous Vehicle Navigation Algorithm," *Proc. SPIE*, v. 485, Arlington, Virginia, 1984.
9. Keirsey, D., "Autonomous Vehicle Control Using AI Techniques," *IEEE Transactions On Software Engineering*, v. SE-11 , no. 9, Sep 1985.
10. Applications of Artificial Intelligence , Meystel, A. and Koch, E., "Computation Simulation of Autonomous Vehicle Navigation," *Proc. SPIE*, v. 485, Arlington, Virginia, 1984.
11. Martin Marietta Astronautics Group, *The Autonomous Land Vehicle Program,* Unpublished Document, Martin Marietta Inc., P.O. Box 179, Denver, Colorado 80201, December 1985.
12. Martin Marietta Astronautics Group, *The Autonomous Land Vehicle Program Seventh Quarterly Report,* Distribution limited to DOD Components only; Critical Technology, pp. 1-6, Commander and Director, U.S. Army Engineer Topographic Laboratories,, Fort Belvoir, Virginia 22060-5546, November 30 1987.
13. Director, Development Center, "Troop Leading Procedure," in *FMFM 6-5*, pp. 8-10, Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402, 1974.
14. Silicon Graphics, Inc., "IRIS System Overview," *IRIS Users Guide*, Mountain View, California, 1986.

15. Matthews, Lawrence, Manuel, Glenn, and Krueger, Steve, *AI Hardware Architectures: Current and Anticipated*, Unpublished Document, Texas Instruments Inc., Dallas, Texas, March 1983.

16. Texas Instruments Inc., "Technical Summary," *Texas Instruments Explorer Technical Reference*, Texas Instruments Inc., Austin, Texas, 1985.

17. Symbolics Inc., "Hardware Overview," *Symbolics Technical Summary*, Cambridge, Massachusetts, 1984.

18. Electronic Systems Engineering Series, Halsall, F., "6.2 The ISO Reference Model," in *Introduction to Data Communications and Computer Networks*, pp. 137-175, Addison-Wesley Publishing Company Inc., Reading, Massachusetts, 1985.

19. Hunter, Bruce H., "Introduction: What is C ?," in *Understanding C*, pp. 3-13, CYBEX Computer Books, Berkeley, California, 1984.

20. MacLennan, Bruce J., "List Processing: LISP," in *Principles of Programming Languages*, pp. 411-438, CBS College Publishing, New York, New York, 1987.

21. Steele, G. L., "Introduction," in *Common Lisp: The Language*, pp. 3-17, Digital Press, Hanover, Massachusetts, 1984.

22. Baker, M. P. and Hearn, D., and Bromley, H. I., "Chapter 2 What's A Flavor ?," in *Lisplore: Guide to Programming the Lisp Machine*, pp. 17-32, Kluwer Academic Publishers, Boston, Massachusetts, 1986.

23. Intellicorp, "Introduction," in *KEE Software Development System User's Manual*, Intellicorp, Mountain View, California, 1986.

24. MacLennan, Bruce J., "Logic Programming: PROLOG," in *Principles of Programming Languages*, pp. 485-540, CBS College Publishing, New York, New York, 1987.

25. 2nd Edition, Clocksin, W. F. and Mellish, C. S., "Tutorial Introduction," in *Programming in Prolog*, pp. 2-3, Springer-Verlag, New York, 1984.

26. Symbolics Inc., "Network Protocols," *Networks*, Cambridge, Massachusetts, 1984.

27. Frank, A. A. and McGhee, R. B., "Some Considerations Relating to the Design of Autopilots for Legged Vehicles," *Journal of Terramechanics*, v. 6 , no. 1 , pp. 23-35, Great Britain, 1969.

28. Fu, K. S., Gonzalez, R. C., and Lee, C. S. G., "Robot Arm Kinematics," in *ROBOTICS: Control, Sensing, Vision and Intelligence*, pp. 12-76, McGraw-Hill Book Company, New York, New York, 1987.

29. Winston, P. H. and Horn, B. K. P., "Rule Based Systems," in *Lisp 2nd Edition*, pp. 243-311, McGraw-Hill, Cambridge, Massachusetts, 1985.

30. Rowe, N. C., *Artificial Intelligence Through Prolog*, pp. 137-149, Prentice Hall, Englewood Cliffs, New Jersey 07632, 1988.

31. Barrow, T. H., *Distributed Computer Communications in Support of Real-Time Visual Simulations,* M. S. Thesis, Naval Postgraduate School, Monterey, California, June 1988.

32. Charniak, E. and McDermott, D. V., "Parsing Language," in *Introduction To Artificial Intelligence*, pp. 170-246, Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

APPENDIX A - THE PILOT CONTROLLER

TANKCONTROL.LISP

```
;;;; system variables

(defvar *battle*          nil)            ;;; the tcp sesseion with iris
(defvar *deltaspeed*       1.0)             ;;; 1/2 the delta tank velocity
(defvar *speed-factor*     2.0)
(defvar *deltacourse*     10.0)             ;;; course change in degree's

(defvar *tank-1-assertions* nil)            ;;; knowledge it's born with
(defvar *tank-2-assertions* nil)
(defvar *tank-3-assertions* nil)
(defvar *tank-4-assertions* nil)
(defvar *tank-5-assertions* nil)

(defvar *formation-rules*  nil)     ;;; rules about a particular formation
(defvar *proper-distance*  nil)     ;;; the proper distance  to maintain
(defvar *tanks-in-unit*    5)     ;;; number of tanks in unit
(defvar *next-time*       20.0)    ;;; approx interval between think times
(defvar *last-x*          0.0)     ;;;
(defvar *last-y*          0.0)     ;;;
(defvar *proper-interval*  20.0)     ;;; interval in meters between tanks
(defvar *seconds-per-frame* 0.2)
;;;; the unit loop

(defun relative-time (distance speed)
 (/ distance  speed))

(defun calculate-relative-time (c &aux veh x y spd time)
 (progn
  (setq veh (send *battle* :lookat c))
  (setq x (getx veh))
  (setq y (getz veh))
  (setq spd (getspd veh))
  (setq time (relative-time (coorddistance *last-x* *last-y* x y) spd))
  (setq *last-x* x)
  (setq *last-y* y)
  time))

(defun start-the-battle (x clockvehicle &aux veh)
  (setq veh (send *battle* :lookat clockvehicle))
  (setq *last-x* (getx veh))
  (setq *last-y* (gety veh))
  (setq *next-time* (calculate-relative-time clockvehicle))
  (loopfor *tanks-in-unit* 1 x
   (progn
    (setq *next-time* (calculate-relative-time clockvehicle))
    (assume-control 1)
;   (assume-control 2)
;   (assume-control 3)
;   (assume-control 4)
:   (assume-control 5)
    )))

;;;; this is the highlevel control loop for a tank

(defun assume-control (tank &aux observations)
  (progn (terpri)
         (princ " Tank #") (princ tank) (princ " now concious ")
         (terpri)
;;;; can switch to different vehicle to view from
;;;;     (send *battle* :viewer tank)
         (setq observations (visio tank))
         (forward-chain (append observations
                                (get_tank_knowledge tank))
                        *formation-rules*)
         (taskexecuter (reverse tasklist))))

;;;; this gets the individual characteristics of a specific tank

(defun get_tank_knowledge (tank)
  (cond
   ((equal tank 1) *tank-1-assertions*)
   ((equal tank 2) *tank-2-assertions*)
   ((equal tank 3) *tank-3-assertions*)
   ((equal tank 4) *tank-4-assertions*)
   ((equal tank 5) *tank-5-assertions*)
   (t  '"tank doesn't exist")))


;;; this functions uses side effects to load in the assertions and rules of each tank

(DEFUN gettanks (FILE)
  (BLOCK gettanks
       (LET ((FP (OPEN FILE :DIRECTION :INPUT)))
          (setq *tank-1-assertions* (if fp (read fp) nil))
         (setq *tank-2-assertions* (if fp (read fp) nil))
          (setq *tank-3-assertions* (if fp (read fp) nil))
          (setq *tank-4-assertions* (if fp (read fp) nil))
          (setq *tank-5-assertions* (if fp (read fp) nil))
         (setq *formation-rules*   (if fp (read fp) nil))
         (PROGN (CLOSE FP) 't))))
```

APPENDIX B - THE VISION SIMULATOR

VISION.LISP

```
;;;;; significant observations ;;;;;;;

(defun toofar (x y)
  `(,x is too far from ,y))

(defun tooclose (x y)
  `(,x is too close to ,y))

(defun rightof (x y)
  `(,x is right of ,y))

(defun leftof (x y)
  `(,x is left of ,y))

(defun forward (x y)
  `(,x is ahead of ,y))

(defun behind (x y)
  `(,x is behind ,y))

(defun aheadof (x y)
  `(,x is ahead of ,y))

(defun online (x y)
  `(,x is online with ,y))

(defun oncourse (x y)
  `(,x is on course with ,y))

(defun oncolumn (x y)
  `(,x is on column with ,y))

(defun offcourse (x y compass degrees)
  `(,x is off course with ,y by ,degrees ,compass))

(defun samespeed (x y)
  `(,x is same speed as ,y))

(defun slower (x y speed)
  `(,x is slower than ,y by ,speed))

(defun faster (x y speed)
  `(,x is faster than ,y by ,speed))

;;;;; significant future observations ;;;;;;;

(defun wtoofar (x y)
  `(,x will be too far from ,y))
```

```
(defun wtooclose (x y)
  `(,x will be too close to ,y))

(defun wrightof (x y)
  `(,x will be right of ,y))

(defun wleftof (x y)
  `(,x will be left of ,y))

(defun wforward (x y)
  `(,x will be ahead of ,y))

(defun wbehind (x y)
  `(,x will be behind ,y))

(defun waheadof (x y)
  `(,x will be ahead of ,y))

(defun stopped (x)
  `(,x is stopped))

(defun backingup (x)
  `(,x is backing up))

(defun metersfrom (axis x y d)
  `(,x is ,d meters from ,y on ,axis axis))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;; significant tests to make the observations ;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun distancefrom (x1 y1 x2 y2) (fix (sqrt (+ (si::sqr (- x1 x2)) (si::sqr (- y1 y2))))) )


;;;; some very rule of thumb directions

(defun northp (x) (and (>= x 315) (<= x 45)))
(defun southp (x) (and (<= x 225) (>= x 135)))
(defun eastp  (x) (and (>= x 45) (<= x 135)))
(defun westp  (x) (and (>= x 225) (<= x 315)))
(defun nwp    (x) (and (>= x 270) (<= x 360)))
(defun swp    (x) (and (>= x 180) (<= x 270)))
(defun nep    (x) (and (>= x 0)  (<= x 90)))
(defun sep    (x) (and (>= x 90) (<= x 180)))

(defun sqr (x) (* x x))

(defun pt-to-pt-distance (t1 t2)
  (sqrt (+ (sqr (- (getx t1) (getx t2)))
          (sqr (- (getz t1) (getz t2))))))

(defun coorddistance (x1 z1 x2 z2)
  (sqrt (+ (sqr (- x1 x2))
          (sqr (- z1 z2)))))
```

```
(defun distance-will-travel (spd time)
  (* spd time))

(defun approx-x (x s time d)
  (+ x  (* s time (sin (radian d)))))

(defun approx-z (z s time d)
  (+ z  (* s time (cos (radian d)))))


(defun relative-position-matrix (x1 z1 x2 z2 rotation-angle)
  (m* (make-vector
       (make-vector
        (+ (- 0 x1)  x2)

        (+ (- 0 z1)  z2)
        0))
     (rzt rotation-angle)))

;;;; sort the objects from near to far

(defun get-closest (t1 view &aux closest)
  (setq closest (car view))
  (dolist (x (cdr view) closest)
    (if (< (pt-to-pt-distance t1 x) (pt-to-pt-distance t1 closest))
        (setq closest x))))

(defun sort-view (t1 v)
  (cond ((null v) nil)
        (t (cons (get-closest t1 v)
               (sort-view t1 (delete (get-closest t1 v) v :test 'equal))))))

(defun visio  (tank)
  (let* ((returned (send *battle* :vision tank))              ;look inside iris
         (*tank* (car returned))                          ;the tank himself
         (*vision* (sort-view *tank* (cdr returned)))           ;objects in his line-of-sight
         (*observations* nil)                             ;what does he see
        (n1  (getname *tank*))                            ;tank's name
        (x1  (getx *tank*))                               ;tank's x coord now
        (z1  (getz *tank*))                               ;tank's z coord now
        (s1  (getspd *tank*))
         (d1  (getdir *tank*))
        (rotation-angle (- 360.0 d1)))
   (cond ((null *vision*) *observations*)                  ;no objects seen, nothing to observe

         (t
           (if (= 0 (fix s1)) (setq *observations* (cons (stopped n1) *observations*)))

           (setq *observations* (cons '(,n1 speed ,s1) *observations*))

          (catch 'exit
            (dolist (object *vision* *observations*)          ;do for all objects seen
             (let*
               ((n2 (getname object))                     ;object's    name
             (x2 (getx object))                    ;          x
             (z2 (getz object))                    ;          y
            (s2 (getspd object))
             (d2 (getdir object))

             (relative-position                    ;relative to x1 z1
              (relative-position-matrix x1 z1 x2 z2 rotation-angle))

             (x2rel (matrix-aref relative-position 0 0))          ;relative x now for object
             (z2rel (matrix-aref relative-position 0 1))          ;  "    z  "
             (d2rel (float (rem (fix (+ d2 rotation-angle)) 360)))
             (x2nxt (approx-x x2rel s2 *next-time* d2rel))  ;rel x
             (z2nxt (- 0 (- (distance-will-travel s1 *next-time*)
                        (approx-z z2rel s2 *next-time* d2rel)))) ;rel z
             )
;;;          (catch 'exit
             (progn
 (terpri)
              (princ "name   = ") (princ n1)
 (terpri)
            (princ "x1     = ") (princ x1)
 (terpri)
            (princ "z1     = ") (princ z1)
 (terpri)
            (princ "speed  = ") (princ s1)
 (terpri)
            (princ "direct  = ") (princ d1)
 (terpri)
              (princ "r-angle = ") (princ rotation-angle)
 (terpri)
              (princ "name   = ") (princ n2)
 (terpri)
              (princ "x2     = ") (princ x2)
 (terpri)
              (princ "z2     = ") (princ z2)
 (terpri)
            (princ "speed  = ") (princ s2)
 (terpri)
              (princ "direct  = ") (princ d2)
 (terpri)
            (princ "reldir  = ") (princ d2rel)
 (terpri)
              (princ "x2rel   = ") (princ x2rel)
 (terpri)
              (princ "z2rel   = ") (princ z2rel)
 (terpri)
              (princ "x2nxt   = ") (princ x2nxt)
 (terpri)
              (princ "z2nxt   = ") (princ z2nxt)
 (terpri)
            (princ "distance= ") (princ (coorddistance x2rel z2rel x2nxt z2nxt))
 (terpri)
            (princ "*next-time*  == ") (princ *next-time*)
 (terpri)
```

```
;;;; check future lateral alignment

(if (and (<= *proper-interval* x2nxt)
         (<= *proper-interval* x2rel))
    (setq *observations* (cons (oncolumn n1 n2) *observations*)))

  (if (< 0.0 x2nxt)
     (setq *observations* (cons (wleftof n1 n2)  *observations*)))
  (if (> 0.0 x2nxt)
     (setq *observations* (cons (wrightof n1 n2) *observations*)))
  (if (< 0.0 x2rel)
     (setq *observations* (cons (leftof n1 n2)  *observations*)))
  (if (> 0.0 x2rel)
     (setq *observations* (cons (rightof n1 n2) *observations*)))


  ;;;; check future vertical alignment


(if (and (< (- 0 *proper-interval*) z2nxt)
         (> *proper-interval* z2nxt))
    (setq *observations* (cons (online n1 n2) *observations*))

    (cond ((> 0.0  z2nxt)
          (setq *observations* (cons (waheadof n1 n2)  *observations*)))
         ((< 0.0 z2nxt)
          (setq *observations* (cons (wbehind n1 n2) *observations*))))

    )


  ;;;; check course alignment

  (if (or (>= 1.00 d2rel) (>= 1.0 (- 360 d2rel)))
     (setq *observations* (cons (oncourse n1 n2) *observations*))
     (setq *observations* (cons
                         (if (> d2rel 180.0)
                         (offcourse n1 n2 'west (- 360.0 d2rel))
                         (offcourse n1 n2 'east d2rel))
                         *observations*)))

  ;;;; check speed

  (if (= 0 (fix s2)) (setq *observations* (cons (stopped n2) *observations*)))

  (if (> 1 (fix (abs (- s1 s2))))
     (setq *observations* (cons (samespeed n1 n2) *observations*))
     (cond
       ((< (fix s1) (fix s2))
        (setq *observations* (cons (slower n1 n2 (- s2 s1))
                              *observations*)))
       ((> (fix s1) (fix s2))
        (setq *observations* (cons (faster n1 n2 (- s1 s2))
                              *observations*)))))
```

```
(setq *observations* (cons '(,n2 speed ,s2) *observations*))

;;;; check future interval

(if (< (coorddistance 0 0 x2nxt z2nxt) *proper-interval*)
   (progn
     (setq *observations* (cons (wtooclose n1 n2)  *observations*))
      (throw 'exit *observations*)))

(if (> (coorddistance 0 0 x2nxt z2nxt) *proper-interval*)
   (setq *observations* (cons (wtoofar n1 n2) *observations*)))


(setq *observations* (cons (metersfrom 'z n1 n2 (abs z2nxt))
                           *observations*))

(setq *observations* (cons (metersfrom 'x n1 n2 (abs x2nxt))
                           *observations*))
;;;; check interval

(if (< (coorddistance 0 0 x2rel z2rel) *proper-interval*)
   (progn
     (setq *observations* (cons (tooclose n1 n2)  *observations*))
      (throw 'exit *observations*)))

(if (> (coorddistance 0 0 x2rel z2rel) *proper-interval*)
   (setq *observations* (cons (toofar n1 n2) *observations*)))


  )))
)))))
```

APPENDIX C - THE TASK GENERATOR

TASKGENERATOR.LISP

```
;;;assertions are represented as lists of atoms
;;;the database will be a list of assertions
;;;this can be built later to represent worlds or contexts (same same)


(defvar assertions nil)          ;fact database used by the forward chainer
(defvar rules nil)               ;rule database used by the forward chainer
(defvar rules-used-list nil)     ;history of conclusions reached by forward chainer
                                 ;for how and why questions
(defvar tasklist nil)            ;the task list generated by the thinker

 ; develop the matcher  - matchs a pattern and a datum, both of
 ;                 which are lists.
 ; datum   : hypotheses or facts, assertions about some real or supposed world
 ; pattern : goals and conditions (or rules)
 ;         '?' = wild card one element in pattern
 ;             an atom variable
 ;         '+' = wild card variable number of elements in pattern
 ;             a list variable i.e. add this and its value list to var-val
 ;         '>' = add the var and value atom to the var-val list
 ;         '<' = replace this var with the val of the var-val pair
 ;             inserted into the var-val list by a previous '>'
 ;         '>* = add the list to the var-val list
 ;         '<*'= replace this var with val of a previous '+'

 ; this pulls out the pattern symbol

(defun pattern-indicator (l) (if (listp l) (car l ) l))

 ; this pulls out the pattern variable

(defun pattern-variable  (l) (if (listp l) (cadr l) l))

 ; this adds a var-val pair to the remembered list

(defun addvarval (variable item assoc-list)
 (append (if (equal assoc-list t) nil assoc-list) (list (list variable item))))

 ; this add a var-listval pair to the remembered list

(defun addvarlval (variable item assoc-list)
  (cond ((null assoc-list) (list (list variable (list item))))
        ((equal variable (caar assoc-list))
         (cons (list variable (append (cadar assoc-list) (list item)))
            (cdr assoc-list)))
        (t (cons (car assoc-list)
             (addvarlval variable item (cdr assoc-list))))))
```

```
; this gets the value of a variable stored in the remembered list

(defun getvarval (variable assoc-list)
  (cadr (assoc variable assoc-list)))

; next three routines 'type' check values --- see end of match routine
; this gets the restrict symbol

(defun restriction-indicator (pattern-item) (cadr pattern-item))

; this gets the predicates that are eval'd as constraints

(defun restriction-predicates (pattern-item) (cddr pattern-item))

; this uses the predicates to test the restrictions

(defun test (predicates argument)
  (cond ((null predicates) t)
       ((funcall (car predicates) argument)
        (test (cdr predicates) argument))
       (t nil)))

;*******************************************************************************
; start the matcher --- this is the heart of the inference engine
; p = pattern, d = data, assignments = association list of var-value pairs
;*******************************************************************************

(defun match ( p d assignments)

       ;pattern matches datum ... return either t or variable assignments
       ;or new assertions created by forward chaining

  (cond  ((and (null p) (null d))                 ;p and d both empty'd?
        (cond ((null assignments) t)               ;success
             (t assignments)))

       ;pattern doesn't match ... return false truth value
        ((or (null p) (null d)) nil)              ;one list shorter?

       ;wild card pattern or matching elements same
        ((or (equal (car p) '?)                   ;pattern element wild?
           (equal (car p) (car d)))               ;first elements same?
           (match (cdr p) (cdr d) assignments ))   ;match

       ;wild card try match on elements or wildcard it a while
       ;these indicate list variables
        ((equal (car p) '+)                       ;match + pattern
           (or (match (cdr p) (cdr d) assignments)
              (match p (cdr d) assignments)))

       ;error in a match .... die
        ((atom (car p)) nil)                      ;losing atom

       ;atomic variables add var-value pair to list after match
  ((equal (pattern-indicator (car p)) '>)        ;match > variable
     (match (cdr p) (cdr d)
          (addvarval (pattern-variable (car p))
                 (car d)
                 assignments)))

  ;this is a list variable indicator , add the list to the var val list
  ((equal (pattern-indicator (car p)) '>*)       ;match >* variable
     (let ((new-assignments (addvarlval (pattern-variable (car p))
                              (car d)
                              assignments)))

       (or (match (cdr p) (cdr d) new-assignments)
          (match   p   (cdr d) new-assignments))))

  ;make a value substitution at this point using the var val list
  ;an then forward chain again using the substituted value
  ((equal (pattern-indicator (car p)) '<)       ;substitute variable
     (match (cons (getvarval (pattern-variable (car p))
                        assignments)
              (cdr p))
            d
            assignments))

  ;make a list substitution at this point and match again
  ((equal (pattern-indicator (car p)) '<*)
     (match (append (getvarval (pattern-variable (car p))
                          assignments)
               (cdr p))
            d
            assignments))

  ;the idea is that the corresponding position in the datum
  ;must be occupied by an atom that satisfies all of the predicates
  ;listed in the restriction  i.e.,
  ;     (restrict  a ?variable or >variable pred1 ... predn)

  ((and (equal (pattern-indicator (car p))       ;match restriction
            'restrict)
        (equal (restriction-indicator (car p)) '?)
        (test (restriction-predicates (car p)) (car d)))
   (match (cdr p) (cdr d) assignments))
))
;*************************************************************************
;************************** end the matcher ****************************
;*************************************************************************

;tasks are represented as high level lisp function calls


;this procedure adds a task to the tasklist or a new assertion to the database

(defun remember (new)
  (if (equal (car new) 'task)                    ;is it a task?
    (cond ((member (cdr new) tasklist :test 'equal) nil)   ;if so add it to tasklist
         (t (setq tasklist (cons (cdr new) tasklist)))))

  (cond ((member new assertions :test 'equal) nil)  ;if present, no action
      (t (setq assertions (cons new assertions))  ;if not add it
        new)))                                   ;return the new assertion

;this procedure finds all assertions that match a given pattern

(defun recall (p) (recall1 p assertions))          ;assertions is free
(defun recall1 (p a)
 (cond ((null a) nil)
      ((match p (car a) nil)                   ;a match?
         (cons (car a) (recall1 p (cdr a))))    ;add it to list of founds
      (t (recall1 p (cdr a)))))                 ;if not, next assertion

;a problem solver is doing forward chaining if it starts with a collection
;of assertions and tries all available rules over and over, adding new
;assertions as it goes until no rules apply.

;implement this in streams of assertions

(defun combine-streams (s1 s2) (append s1 s2))
(defun add-to-stream (e s) (cons e s))
(defun first-of-stream (s) (car s))
(defun rest-of-stream (s) (cdr s))
(defun empty-stream-p (s) (null s))
(defun make-empty-stream () nil)

;given a pattern and an initial input association list for the matcher
;create another association list of matchings

(defun filter-assertions (p i)
  (do ((a assertions (cdr a))
      (s (make-empty-stream)))
     ((null a) s)
     (let ((n (match p (car a) i)))
     (cond (n (setq s (add-to-stream n s)))))))

;combine the results of many applications of filter-assertions to
;create another stream of the results

(defun filter-a-stream (p a)
 (cond ((empty-stream-p a) (make-empty-stream))
      (t (combine-streams
           (filter-assertions p (first-of-stream a))
           (filter-a-stream p (rest-of-stream  a))))))

;create means of using filter-a-stream once for each antecedent (precedent)
;passing the output of one use to the input of the next

(defun cascade-thru (p a)
 (cond ((null p) a)
      (t (filter-a-stream (car p) (cascade-thru (cdr p) a)))))
```

```
; feed-to-actions feeds the a-list streams of filterd ifs to s-actions
; feed-to-actions also combines the resulting action streams into a single one

 (defun feed-to-actions (rule-name actions a)
  (cond ((empty-stream-p a) (make-empty-stream))
       (t (combine-streams (s-actions      rule-name
                                actions
                                (first-of-stream a))
                       (feed-to-actions rule-name
                                actions
                                (rest-of-stream a))))))

; s-actions replaces pattern variables in the action with values, tries
; to add the resulting assertion to the data, and contributes to new action
; streams

 (defun s-actions (rule-name actions a)
  (do ((actx actions (cdr actx))
      (as (make-empty-stream)))
     ((null actx) as)
     (let* ((acty (replace-var (car actx) a))
          (act (if  (and (listp (car actx))
                      (listp (caar actx))
                      (equal '<* (caaar actx)))
                 (append (car acty) (cdr acty))
                  acty))
          )
       (cond ((remember act)
            (print `(rule ,rule-name says ,@act))
            (setq as (add-to-stream act as)))))))

; replace variable names with values

 (defun replace-var (s a)
  (cond ((atom s) s)
       ((equal (car s) '<)
        (cadr (assoc (pattern-variable s) a)))
       ((equal (car s) '<*)
        (cadr (assoc (pattern-variable s) a)))
       (t   (cons  (replace-var (car s) a)
               (replace-var (cdr s) a)) )))

; replacement procedure

(defun repl (s a)
  (cond ((null s) nil)
       ((atom s) s)
       ((or (equal (pattern-indicator s) '< )
          (equal (pattern-indicator s) '> )
          (equal (pattern-indicator s) '<* )
          (equal (pattern-indicator s) '>*))
        (cadr (assoc (pattern-variable s) a ))
       )
       (t   (cons (repl (car s) a)
                (repl (cdr s) a)))))

;extract list of patterns from a rule  record rule if it was used
;recorded in global rules-used-list

 (defun use-rule (rule)
  (let* ((rule-name (cadr rule))
        (ifs      (reverse (cdr (caddr rule))))
        (thens    (cdr    (cadddr rule)))
        (a (cascade-thru ifs (add-to-stream nil (make-empty-stream))))
        (a-stream (feed-to-actions rule-name thens a )))
  (cond ((not (empty-stream-p a-stream))
        (rules-used rule-name ifs thens a) t ))))

 (defun rules-used (rule-name ifs thens a)
 (cond ((empty-stream-p a) t)
     (t  (setq rules-used-list
            (cons (list rule-name (repl ifs (first-of-stream a))
                               (repl thens (first-of-stream a)))
                  rules-used-list))
         (rules-used rule-name ifs thens (rest-of-stream a)))))

; used rule predicate "have you used rule ...?
; useful to trim excess rules from the rule database

(defun rulep (rule) (cond ((assoc rule rules-used-list) t) (t nil)))

; "how did you deduce that ... ?
; prints the assertions that allowed the deduction of the argument

(defun how (fact) (how1 fact rules-used-list nil))

(defun how1 (fact possible success)
  (cond ((null possible) (cond ( success      t)
                        ((recall fact) (print '(,@fact was given)) t)
                        (t (print '(,@fact is not established)) nil)))
      ((member fact (caddr (car possible)) :test 'equal)
          (print '(,@fact because))
          (mapcar #'(lambda (a) (print a)) (cadr (car possible)))
          (how1 fact (cdr possible) t))
      (t    (how1 fact (cdr possible) success))))

; explaintasks explains the how the tank identified its tasks to accomplish

(defun explain (tasks)
  (cond ((null tasks) t)
       (t (terpri) (how (cons 'task (car tasks))) (explain (cdr tasks)))))

; "why did you need that assertion ... ?
; why prints the assertions that depend on its fact

(defun why (fact) (why1 fact rules-used-list nil))

(defun why1 (fact possible success)
(cond ((null possible) (cond (success t)
                        (t (print '(,@fact was not used)) nil)))
    ((member fact (cadr (car possible)) :test 'equal)
        (print '(,@fact is needed to show))
        (mapcar #'(lambda (a) (print a))
                            (caddr (car possible)))
        (why1 fact (cdr possible) t))
    (t    (why1 fact (cdr possible) success))))

; forward-chain steps thru rule list until it finds a rule that produces a
; new assertion whereupon it starts over at beginning of rule list.
; fc stops when it fails to find a new assertion with any rule
; returns the rules-used-list or history of the forward-chain

(defun forward-chain (tankfacts tankrules)
  (setq rules-used-list (make-empty-stream))
  (setq tasklist (make-empty-stream))
  (setq assertions  tankfacts)
  (setq rules tankrules)
  (do ((rules-to-try rules (cdr rules-to-try))
      (progress-made nil))
      ((null rules-to-try) progress-made)
      (cond ((use-rule (car rules-to-try))
          (setq rules-to-trv rules)
          (setq progress-made t))))
   (explain (reverse tasklist)))
```

APPENDIX D - THE TASK EXECUTOR

TASKEXECUTOR.LISP

```
(defun taskexecuter (tasklist)
 (cond ((null tasklist) t)
       (t (eval (car tasklist)) (taskexecuter (cdr tasklist)))))

;;;;; direction changes ;;;;;;:·:

(defun Change-direction-to (tank direction)
 (send *battle* :task-exec tank "Q" direction))

(defun Move-right (tank delta)
   (send *battle* :task-exec tank "D" delta))

(defun Move-left (tank delta)
  (send *battle* :task-exec tank "D" (- 0 delta)))

(defun Turn-right (tank)
 (send *battle* :task-exec tank "D" 90.0))

(defun turn-left  (tank)
 (send *battle* :task-exec tank "D" -90.0))

(defun Move-to-right (tank)
 (move-right tank 20.0)
 (move-left tank 20.0))

(defun Move-to-left (tank)
 (move-left tank 20.0)
 (move-right tank 20.0))

;;;;;; speed changes  ;;;;;;;;;;;

(defun Change-speed-to (tank speed)
 (send *battle* :task-exec tank "X" speed))

(defun Back-up (tank)
 (send *battle* :task-exec tank "X" -2.0))

(defun Stop (tank)
 (send *battle* :task-exec tank "X" 0.0))

(defun Surge (tank)
 (send *battle* :task-exec tank "X" 25.0))

(defun Increase-speed (tank delta)
 (send *battle* :task-exec tank "S" delta))

(defun Decrease-speed (tank delta)
 (send *battle* :task-exec tank "S" (- 0 delta)))


;;;;;; do commands concurrantly for whole formation

(defmacro parade (veh cmd)
 `(loopfor *parade* 1 (1+ ,veh) (funcall ',cmd *parade*)))

(defun gettime (tank)
 (send *battle* :clock tank))

(defun getframerate (tank)
 (send *battle* :frame-interval tank))

(defun getloopcnt (tank)
 (send *battle* :frame-count tank))
```

APPENDIX E - THE TASK INTERFACE LAYER

TANKTALK.LISP

```
;;; definitions:
;;;
;;;   object: "n"             name:       character "1" .. "5"
;;;          x               x coordinate: real
;;;          y               y coordinate: real
;;;          z               z coordinate: real
;;;          spd              speed:      real   speed of vehicle -10.00 to 25.00
;;;          dir             direction:   real   compass dir in degrees from GN
;;;
;;;   in lisp  ("n" (x y z spd dir))

(defun makeobj (n x y z spd dir)
  (list n (list x y z spd dir)))

(defun getname (o) (car o))

(defun getposition (o) (cadr o))

(defun getx (o) (car (getposition o)))

(defun gety (o) (cadr (getposition o)))

(defun getz (o) (caddr (getposition o)))

(defun getspd (o) (cadddr (getposition o)))

(defun getdir (o) (car (cddddr (getposition o))))

;;; vision module
;;; ti sends:  "V"                 for vision
;;;           "n"              char "1" .. "5" for tank doing the look see

;;; iris then does its stuff and returns following msg

;;; iris sends: "V"                 for vision
;;;           #n               # of objects to follow
;;;           object1 .. objectn     the objects within a 100m radius of the tank
;;;           object              updated object whose vision the following
;;;                            stream represents


;;; get an object in graphics environment (defined as above)

(defmethod (conversation-with-iris :object) ()
  (makeobj
    (send self :get-iris)
    (send self :get-iris)
  (send self :get-iris)
  (send self :get-iris)
  (send self :get-iris)
  (send self :get-iris)))

;;; vision returns a list of objects in the tank's field of vision (100m radius)
;;; this is effectively an association list

(defmethod (conversation-with-iris :vision) (tank)
 (let ((field nil)
       (n-objects 0))
    (progn (send self :put-iris "V")
           (send self :put-iris tank)
            (if (equal "V" (send self :get-iris))
              (progn (setq n-objects (send self :get-iris))
                     (dotimes (x n-objects field) (setq field (cons (send self :object) field))))
                (progn (print "iris did not respond to the vision command sent from ")
                       (princ "tank ") (princ tank))))))


;;; task execution
;;; ti sends:         "T"       standby for following task cmd
;;;                  "n"       object name of requestor "1" .. "5"
;;;                  "t"       task to exec "S" "D" "E" "T" ...
;;;                  r         real number delta of task to accomplish
;;; iris returns      "t"       task exec'd  "S" "D" "E" "T" ...
;;;                  object    changed object (one who is executing task)
;;;
;;; tasks implemented:
;;;                  "S"    change speed by delta
;;;                  "D"    change direction by delta
;;;                  "E"    elevate gun by delta
;;;                  "T"    traverse gun by delta
;;;
;;; tasks to implement:
;;;                  "X"    stop the tank
;;;

(defmethod (conversation-with-iris :task-exec) (tank task delta)
 (let ((object nil))
    (progn (send self :put-iris "T")
           (send self :put-iris tank)
           (send self :put-iris task)
           (send self :put-iris delta)
           (if (equal task (send self :get-iris))
             (progn (setq object (send self :object))
                    (if (equal tank  (getname object))
                      object
                      (progn (print "iris did not exec task for specified tank")
                             (princ "tank requesting        ") (print tank)
                             (princ "tank received from iris ") (print object))))
               (progn (print "iris did not respond to a task-exec request message for")
                      (princ "tank ") (print tank)
                      (princ "task ") (print task)
                (send self :object))))))

;;; change the iris out of windshield view to the specified tank

(defmethod (conversation-with-iris :viewer) (tank)
 (let ((object nil))
   (progn (send self :put-iris "P")
          (send self :put-iris tank)
          (if (equal "P" (send self :get-iris))
             (progn (setq object (send self :object))
                    (if (equal tank (getname object))
                        object
                     (progn (print "iris did not change view to specified tank")
                               (princ "tank requesting        ") (print tank)
                               (princ "tank received from iris ") (print object))))
             (progn (print "iris did not respond to a change view request message to")
                   (princ "tank ") (print tank)
                    (send self :object))))))

;;; look at a specific tank or object
;;; not implemented

(defmethod (conversation-with-iris :lookat) (tank)
 (let ((object nil))
    (progn (send self :put-iris "L")
          (send self :put-iris tank)
           (if (equal "L" (send self :get-iris))
             (progn (setq object (send self :object))
                    (if (equal tank (getname object))
                        object
                     (progn (print "iris did look at specified tank")
                               (princ "tank requesting        ") (print tank)
                               (princ "tank received from iris ") (print object))))
             (progn (print "iris did not respond to a lookat request message to")
                   (princ "tank ") (print tank)
                    (send self :object))))))


;;; get the elapsed time from the iris
;;;

(defmethod (conversation-with-iris :clock) (tank)
  (progn (send self :put-iris "C")
        (send self :put-iris tank)
        (if (equal "C" (send self :get-iris))
           (send self :get-iris)
           (progn (print "iris did not return time ")
                 (princ "Tank requesting ")
                 (print tank)))))


;;; get the time interval per frame write
;;; this is the discrete interval between picture updates of the tank
```

```
(defmethod (conversation-with-iris :frame-interval) (tank)
  (progn (send self :put-iris "I")
         (send self :put-iris tank)
         (if (equal "I" (send self :get-iris))
            (send self :get-iris)
            (progn (print "iris did not return time interval")
                   (princ "Tank requesting ")
                   (print tank)))))

;;; get the frame-count or number of times looped thru program from the iris
;;;

(defmethod (conversation-with-iris :frame-count) (tank)
  (progn (send self :put-iris "U")
         (send self :put-iris tank)
         (if (equal "U" (send self :get-iris))
            (send self :get-iris)
            (progn (print "iris did not return loopcount")
                   (princ "Tank requesting ")
                   (print tank)))))
```

APPENDIX F - THE TCP/IP APPLICATION LAYER TI LISP MACHINE

IRISFLAVOR.LISP

```
(defmacro loopfor (var init test exp1 &optional exp2 exp3 exp4 exp5)
  `(prog ()
        (setq ,var ,init)
        tag
        ,exp1
        ,exp2
        ,exp3
        ,exp4
        ,exp5
        (setq ,var (1+ ,var))
        (if (= ,var ,test) (return t) (go tag))))

(defun convert-number-to-string (n)
  (princ-to-string n))

(defun convert-string-to-integer (str &optional (radix 10))
  (do ((j 0 (+ j 1))
       (n 0 (+ (* n radix) (digit-char-p (char str j) radix))))
      ((= j (length str)) n)))

(defun find-period-index (str)
  (catch 'exit
   (dotimes (x (length str) nil)
    (if (equal (char str x) (char "." 0))
        (throw 'exit x)))))

(defun get-leftside-of-real (str &optional (radix 10))
  (do  ((j 0 (1+ j))
        (n 0 (+ (* n radix) (digit-char-p (char str j) radix))))
      ((or (null (digit-char-p (char str j) radix)) (= j (length str)))  n)))

(defun get-rightside-of-real (str &optional (radix 10))
  (do ((index (1+ (find-period-index str)) (1+ index))
       (factor 0.10 (* factor 0.10))
       (n 0.0 (+ n (* factor (digit-char-p (char str index) radix)))))
      ((= index (length str)) n )))

(defun convert-string-to-real (str &optional (radix 10))
  (+ (float (get-leftside-of-real str radix)) (get-rightside-of-real str radix)))


(defvar *tcp-handler1* (send ip::*tcp-handler* :get-port))
(defvar *tcp-handler2* (send ip::*tcp-handler* :get-port))

(defvar *iris1-port1* 1027)                ; this is the send port
(defvar *iris1-port2* 1026)                ; this is the receive port
```

```
(defvar *iris1-address* 3221866502)
(defvar *iris2-address* 3221866504)
(defvar *iris3-address* 3221866505)

(defvar *dest-address*  nil)                ; the tcp-ip or internet address
                                            ; look in network configuration


(defun iris (x)
 (cond ((equal x 1) (setq *dest-address* *iris1-address*))
      ((equal x 3) (setq *dest-address* *iris3-address*))
      (t         (setq *dest-address* *iris2-address*))))


(defflavor conversation-with-iris ((talking-port-number    *iris1-port1*)
                                  (listening-port-number  *iris1-port2*)
                                  (talking-port          *tcp-handler1*)
                                  (listening-port        *tcp-handler2*)
                                   (destination          *dest-address*)
                                 )
                                  ()
                                 :gettable-instance-variables
                                    :settable-instance-variables
                                    :initable-instance-variables)


(defmethod (conversation-with-iris :start-iris) ()
 (progn
  (send talking-port :open
      :active                   ; tcp will begin the procedure to establish
                               ; connection  (default  vs :passive)
      talking-port-number          ; port number of destination host
      destination                 ; machine name or address if blank and
                               ; in :passive mode local machine waits for
                               ; connection
      30)                         ; set max seconds before read request times out


  (send listening-port :open
     :active                              ;;passive
     listening-port-number
     destination
     30)
  '"A conversation with the iris machine has been established"))


(defmethod (conversation-with-iris :reuse-iris) ()
 (setq *tcp-handler1* (send ip::*tcp-handler* :get-port)
       *tcp-handler2* (send ip::*tcp-handler* :get-port)
       talking-port *tcp-handler1*
       listening-port *tcp-handler2*))
```

```
(defmethod (conversation-with-iris :get-iris) ()
 (let* ((typebuffer  " ")
        (lengthbuffer "    ")
       (buffer      " ")
        (buffer-length 1))
  (progn
     (send listening-port :receive
          typebuffer
          buffer-length
          30
          :wait)

     (send listening-port :receive
          lengthbuffer
          4
          30
          :wait)

     (setq buffer-length (convert-string-to-integer lengthbuffer))
     (setq buffer (make-string buffer-length :initial-element (character 32)))

     (send listening-port :receive
          buffer
          buffer-length
          30
          :wait)


     (cond ((equal typebuffer "I") (convert-string-to-integer buffer))
          ((equal typebuffer "R") (convert-string-to-real   buffer))
          ((equal typebuffer "C") buffer)
          (t nil)))))




(defmethod (conversation-with-iris :put-iris) (object)


 (let* ((buffer (cond
              ((equal (type-of object) 'bignum) (convert-number-to-string object))
             ((equal (type-of object) 'fixnum) (convert-number-to-string object))
              ((equal (type-of object) 'float) (convert-number-to-string object))
              ((equal (type-of object) 'string) object)
              (t "error")))

        (buffer-length  (length buffer))

        (typebuffer     (cond ((equal (type-of object) 'bignum) "I")
                             ((equal (type-of object) 'fixnum) "I")
                             ((equal (type-of object) 'float) "R")
                             ((equal (type-of object) 'string) "C")
                             (t "C")))
        (lengthbuffer   (convert-number-to-string buffer-length))
        (*loopvariable* 0))


   (progn
     (send talking-port :send
          typebuffer
          1
          nil
          nil)
      (if (= (length lengthbuffer) 4)
           (send talking-port :send
               lengthbuffer
               4
               nil
               nil)
           (progn
             (loopfor *loopvariable* (length lengthbuffer) 4
                   (send talking-port :send "0" 1 nil nil))
             (send talking-port :send lengthbuffer (length lengthbuffer) nil nil)))
      (send talking-port :send
          buffer
          buffer-length
          t
          nil))))


(defmethod (conversation-with-iris :stop-iris) ()
  (progn (send talking-port :close) (send listening-port :close)))
```

APPENDIX G - THE TCP/IP APPLICATION LAYER

SYMIRIS.LISP

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: USER -*-

; handy macro to have in the send message farthur down


(defmacro loopfor (var init test exp1 &optional exp2 exp3 exp4 exp5)
 `(prog ()
       (setq ,var ,init)
       tag
         ,exp1
         ,exp2
         ,exp3
         ,exp4
         ,exp5
         (setq ,var (1+ ,var))
         (if (= ,var ,test) (return t) (go tag))))


(defun convert-number-to-string (n)
 (princ-to-string n))

(defun convert-string-to-integer (str &optional (radix 10))
 (do ((j 0 (+ j 1))
    (n 0 (+ (* n radix) (digit-char-p (char str j) radix))))
   ((= j (length str)) n)))

(defun find-period-index (str)
 (catch 'exit
  (dotimes (x (length str) nil)
   (if (equal (char str x) (char "." 0))
      (throw 'exit x)))))

(defun get-leftside-of-real (str &optional (radix 10))
 (do  ((j 0 (1+ j))
      (n 0 (+ (* n radix) (digit-char-p (char str j) radix))))
    ((or (null (digit-char-p (char str j) radix)) (= j (length str)))  n)))

(defun get-rightside-of-real (str &optional (radix 10))
 (do ((index (1+ (find-period-index str)) (1+ index))
    (factor 0.10 (* factor 0.10))
    (n 0.0 (+ n (* factor (digit-char-p (char str index) radix)))))
   ((= index (length str)) n )))

(defun convert-string-to-real (str &optional (radix 10))
 (+ (float (get-leftside-of-real str radix)) (get-rightside-of-real str radix)))
```

```
(defvar *iris-port1* 1027)              ; this is the send port
(defvar *iris-port2* 1026)              ; this is the receive port
(defvar *local-talk-port* 1500)           ; this is the local send port
(defvar *local-listen-port* 1501)         , this is the local receive port


(defflavor conversation-with-iris ((talking-port-number    *iris-port1*)
                         (listening-port-number   *iris-port2*)
                          (local-talk-port-number  *local-talk-port*)
                          (local-listen-port-number *local-listen-port*)
                         (talking-stream)
                         (listening-stream)
                          (destination-host-object)
                       )
                        ()
                          :initable-instance-variables)


(defmethod (:init-destination-host conversation-with-iris)
         (name-of-host)
  (setf destination-host-object (net:parse-host name-of-host)))



(defmethod (:start-iris conversation-with-iris) ()
  (setf talking-stream
        (tcp:open-tcp-stream destination-host-object
                         talking-port-number
                         local-talk-port-number))
  (setf listening-stream
        (tcp:open-tcp-stream destination-host-object
                         listening-port-number
                         local-listen-port-number))
  "A conversation with the iris machine has been established")



(defmethod (:reuse-iris conversation-with-iris  )
         ()
  )

: (setq *tcp-handler1* (send ip::*tcp-handler* :get-port)
:       *tcp-handler2* (send ip::*tcp-handler* :get-port)
:       talking-port *tcp-handler1*
:       listening-port *tcp-handler2*))


(defun read-string (stream num-chars)
  (let ((out-string ""))
    (dotimes (i num-chars)
      (setf out-string (string-append out-string (read-char stream)))))
  out-string))


(defmethod (:get-iris conversation-with-iris) ()
 (let* ((typebuffer   " ")
        (lengthbuffer "    ")
       (buffer      " ")
        (buffer-length 1))
   (progn
    (setf typebuffer
          (read-string listening-stream 1))
    (setf lengthbuffer
          (read-string listening-stream 4))
    (setf buffer-length
          (convert-string-to-integer lengthbuffer))
    (setf buffer
          (read-string listening-stream buffer-length))


      (cond ((equal typebuffer "I") (convert-string-to-integer buffer))
            ((equal typebuffer "R") (convert-string-to-real   buffer))
            ((equal typebuffer "C") buffer)
            (t nil)))))


(defvar *step-var* 0)



(defun my-write-string(string stream)
 (let* ((num-chars (length string)))
   (dotimes (i num-chars)
    (write-char (aref string i) stream))))


(defmethod (:put-iris conversation-with-iris)
         (object)

 (let* ((buffer (cond
                  ((equal (type-of object) 'bignum) (convert-number-to-string object))
                 ((equal (type-of object) 'fixnum) (convert-number-to-string object))
                  ((equal (type-of object) 'single-float) (convert-number-to-string object))
                  ((equal (type-of object) 'string) object)
                  (t "error")))

      (buffer-length  (length buffer))

      (typebuffer    (cond ((equal (type-of object) 'bignum) "I")
                           ((equal (type-of object) 'fixnum) "I")
                           ((equal (type-of object) 'single-float) "R")
                           ((equal (type-of object) 'string) "C")
                           (t "C")))
         (lengthbuffer  (convert-number-to-string buffer-length)))

      (progn
       (my-write-string typebuffer talking-stream)
       (send talking-stream :force-output)

         (if (= (length lengthbuffer) 4)
            (write-string lengthbuffer talking-stream)
            (progn
              (loopfor *step-var* (length lengthbuffer) 4
                      (write-string "0" talking-stream))

              (my-write-string lengthbuffer talking-stream)
             ))

         (send talking-stream :force-output)

         (my-write-string buffer talking-stream)
         (send talking-stream :force-output)

         )))

   (defmethod (:stop-iris conversation-with-iris)
          ()
     (progn (send talking-stream :close)
          (send listening-stream :close)))

   (defun select-host (host-name)
     (send talk :init-destination-host host-name))
```

APPENDIX H - THE CHAOS APPLICATION LAYER

CHAOSFLAVOR.LISP

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: USER -*-


(defun convert-number-to-string (n)
 (princ-to-string n))

(defun convert-string-to-integer (str &optional (radix 10))
 (do ((j 0 (+ j 1))
     (n 0 (+ (* n radix) (digit-char-p (char str j) radix))))
    ((= j (length str)) n)))

(defun find-period-index (str)
 (catch 'exit
  (dotimes (x (length str) nil)
   (if (equal (char str x) (char "." 0))
       (throw 'exit x)))))

(defun get-leftside-of-real (str &optional (radix 10))
 (do  ((j 0 (1+ j))
      (n 0 (+ (* n radix) (digit-char-p (char str j) radix))))
    ((or (null (digit-char-p (char str j) radix)) (= j (length str)))  n)))

(defun get-rightside-of-real (str &optional (radix 10))
 (do ((index (1+ (find-period-index str)) (1+ index))
     (factor 0.10 (* factor 0.10))
     (n 0.0 (+ n (* factor (digit-char-p (char str index) radix)))))
    ((= index (length str)) n )))

(defun convert-string-to-real (str &optional (radix 10))
 (+ (float (get-leftside-of-real str radix)) (get-rightside-of-real str radix)))



(defflavor mychaos ((host-name 'sym1)
                    (contact-name "user-chaos")
                    (contact nil)
                    (userstream nil)
                    )
         ()
        :initable-instance-variables)


(defmethod (mychaos :set-host-name)
          (name-of-host)
 (setf host-name name-of-host))

(defmethod (mychaos :set-contact-name) (name)
  (setf contact-name name))

(defmethod (mychaos :set-contact ) (con)
  (setf contact con))

(defmethod (mychaos :set-stream ) (str)
  (setf userstream str))

(defmethod (mychaos :start-user ) (hostname contactname)
  (progn
    (send self :set-host-name hostname)
    (send self :set-contact-name contactname)
    (send self :set-contact (chaos:connect hostname contactname 13 72000))
    (send self :set-stream (chaos:make-stream contact :direction :bidirectional))
    (terpri)
    (princ "host name "   ) (princ host-name)
    (terpri)
    (princ "contact name ") (princ contact-name)
    (terpri)
  "A conversation using chaos has been established"))

(defmethod (mychaos :start-server ) (contactname)
  (progn
    (send self :set-contact-name contactname)
    (send self :set-contact (chaos:listen contactname))
    (chaos:accept contact)
    (send self :set-stream (chaos:make-stream contact :direction :bidirectional))
    (terpri)
    (princ "host name "   ) (princ host-name)
    (terpri)
    (princ "contact name ") (princ contact-name)
    (terpri)
  "A conversation using chaos has been established"))


(defmethod (mychaos :get ) ()
  (let* ((typebuffer   " ")
        (buffer      " ")
         )
    (progn
     (setq typebuffer
           (send userstream :line-in))
     (setq buffer
           (send userstream :line-in))


       (cond ((equal typebuffer "I") (convert-string-to-integer buffer))
             ((equal typebuffer "R") (convert-string-to-real   buffer))
             ((equal typebuffer "C") buffer)
             (t nil)))))



(defmethod (mychaos :put )
          (object)

   (let* ((buffer (cond
                   ((equal (type-of object) 'bignum) (convert-number-to-string object))
                  ((equal (type-of object) 'fixnum) (convert-number-to-string object))
                   ((equal (type-of object) 'float)  (convert-number-to-string object))
                   ((equal (type-of object) 'string) object)
                   (t "error")))

         (typebuffer    (cond ((equal (type-of object) 'bignum) "I")
                               ((equal (type-of object) 'fixnum) "I")
                               ((equal (type-of object) 'float)  "R")
                               ((equal (type-of object) 'string) "C")
                               (t "C")))
         )


     (progn
      (send userstream :line-out typebuffer)
      (send userstream :force-output)

      (send userstream :line-out buffer)
      (send userstream :force-output)
      't
        )))

  (defmethod (mychaos :stop )
          ()
    (send userstream :close :abort))
```

APPENDIX I - RULE BASED STATION KEEPING

LINEFORMATION.LISP

```
;;; begin list of rules of the form
;;; ((rule <rule-name>        ;begin rule
;;;   (if                     ;begin ifs
;;;     ;antecedent rules
;;;     ((> x) expressions .... (> y))
;;;     ((< y) expressions .... (> z))
;;;   )                       ;end ifs
;;;   (then                   ;begin then
;;;     ;precedent or consequents
;;;     ((< z) expressions .... (< x))
;;;     ((< x) expressions .... (< z))
;;;   )                       ;end thens
;;; )                         ;end rule
;;; )

;;; begin list of assertions of the form
;;;(
;;; ( expressions ....)
;;; ( expressions ....)
;;; ....
;;; ....

;;;) ;end  list of assertions

;;; tank-1 assertions
     (
     (vehicle is 1)
     (formation is line)
     (guide vehicle is 2)
     (right vehicle is 2))


;;; tank-2 assertions
     (
     (vehicle is 2)
     (formation is line)
     (guide vehicle is 3)
     (left vehicle is 1)
     (right vehicle is 3))


;;; tank-3 assertions
     (
     (vehicle is 3)
     (formation is line)
     (lead vehicle)
     (left vehicle is 2)
```

```
        (right vehicle is 4))

;;; tank-4 assertions
        (
        (vehicle is 4)
        (formation is line)
        (guide vehicle is 3)
        (left vehicle is 3)
        (right vehicle is 5))

;;; tank-5 assertions
        (
        (vehicle is 5)
        (formation is line)
        (guide vehicle is 4)
        (left vehicle is 4))


;;; tank rules


        ;;;;;;;; look in the future ;;;;;;;;;;;;;;;

(


        (rule go
           (if
              (formation is line)
              (vehicle is (> v))
              (guide vehicle is (> gv))
              ((< v) is stopped)
              ((< v) will be behind (< gv))
              ((< gv) speed (> spd)))
           (then
              (task change-speed-to (< v) (< spd))))


        (rule avoid-collision-to-right
           (if
              (formation is line)
              (vehicle is (> v))
              ((< v) will be too close to (> ov))     ;vision determination
              ((< v) will be left of (< ov)))         ;vision determination
           (then
              (task move-to-left (< v))))             ;task identified


        (rule avoid-collision-to-left
           (if
              (formation is line)
              (vehicle is (> v))
              ((< v) will be too close to (> ov))     ;vision determination
              ((< v) will be right of (< ov)))        ;vision determination
           (then
        (task move-to-right (< v))))          ;task id'd


    (rule avoid-collision-forward
       (if
          (formation is line)
          (vehicle is (> v))
          ((< v) will be too close to (> ov))    ;vision determination
          ((< v) will be behind (< ov)))         ;vision determination
       (then
          (task stop (< v))
          ((< v) is stopped)))                 ; (Decrease-speed veh)


    (rule avoid-collision-rearward
       (if
          (formation is line)
          (vehicle is (> v))
          ((< v) will be too close to (> ov))    ;vision determination
          ((< ov) speed (> os))
          ((< v) will be ahead of (< ov)))        ;vision determination
       (then
          (task change-speed-to (< v) (< os))))


    (rule change-speed
       (if
          (formation is line)
          (vehicle is (> v))
          (guide vehicle is (> gv))
          ((< v) will be behind (< gv))
          ((< v) is on course with (< gv))
          ((< gv) speed (> gs))
          ((< v) is (> vd) meters from (< gv) on z axis))
       (then
          (task change-speed-to (< v) (/ (< vd) *next-time*))
          ((< v) is on line with (< gv))))


    (rule match-speed
       (if
          (formation is line)
          (vehicle is (> v))
          (guide vehicle is (> gv))
          ((< v) is online with (< gv))
          ((< v) is on course with (< gv))
          ((< gv) speed (> gs)))
       (then
          (task change-speed-to (< v) (< gs))))


    (rule close-right
       (if
          (formation is line)
```

```
      (vehicle is (> v))
      (guide vehicle is (> gv))
      ((< v) will be too far from (< gv))   ;vision
      ((< v) will be left of (< gv))        ;vision
   ((< v) is left of (< gv))             ;vision
      (right vehicle is (< gv)))
   (then
      (task Move-to-right (< v))))

(rule close-left
   (if
      (formation is line)
      (vehicle is (> v))
      (guide vehicle is (> gv))
      ((< v) will be too far from (< gv))   ;vision
      ((< v) will be right of (< gv))       ;vision
      ((< v) is right of (< gv))            ;vision
      (left vehicle is (< gv)))
   (then
      (task Move-to-left (< v))))

(rule turn-left
   (if
      (formation is line)
      (vehicle is (> v))
      (guide vehicle is (> gv))
   ((< v) is off course with (< gv) by (> angle) west))   ;vision
   (then
      (task Move-left (< v) (< angle))
      ((< v) is on course with (< gv))))        ;task

(rule turn-right
   (if
      (formation is line)
      (vehicle is (> v))
      (guide vehicle is (> gv))
      ((< v) is off course with (< gv) by (> angle) east)) ;vision
   (then
      (task Move-right (< v) (< angle))
      ((< v) is on course with (< gv))))            ;task

(rule directions
   (if
      (formation is line))
   (then
      (left is opposite of right)
      (right is opposite of left)))

(rule move-into-position-from-wrong-side-of-guide
   (if
      (fromation is line)
      (vehicle is (> v))
      (guide vehicle is (> gv))
      ((< v) will be behind (< gv))
```

```
        ((< v) is on course with (< gv))
        ((< v) will be (> onedir) of (< gv))
        ((< v) is (> onedir) of (< gv))
        ((< onedir) vehicle is (< gv))
        ((< onedir) is opposite of (> otherdir)))
     (then
        (task turn-(< otherdir) (< v))
        (task turn-(< onedir) (< v))))


(rule stop
     (if
        (formation is line)
        (vehicle is (> v))
        (guide vehicle is (> gv))
        ((< gv) is stopped)
        ((< v) is on course with (< gv))
        ((< v) is on line with (< gv)))
    (then
        (task stop (< v))
        ((< v) is stopped)))

 (rule stop
     (if
        (formation is line)
        (vehicle is (> v))
        (guide vehicle is (> gv))
        ((< v) will be ahead of (< gv)))
     (then
        (task stop (< v))
        ((< v) is stopped)))

)
```

APPENDIX J - COORDINATE TRANSFORMATION FUNCTIONS

TANKPOSITION.LISP

```
;;    some useful robotics functions
;;   functions and code commented out represent portability changes from
;;   PC version of lisp (Xlisp 1.7) to TI-Explorer
;;
;;   this version uses arrays


; trigonometric functions of interest

;;;ti has a better pi already
;;;(setq pi 3.14159)

(defun radian (degree) (* degree (/ pi 180.00)))

(defun degree (radian) (/ radian (/ pi 180.00)))

(defun hypotenuse (x y) (sqrt (+ (* x x) (* y y))))

(defun opposite (x r) (sqrt (- (* r r) (* x x))))

(defun adjacent (y r) (opposite y r))

;; the following use information hiding for portability
;; these are changed to current lisp standards for particular machine
;; create a vector

(defun make-vector (&rest x &aux y)
;;(setq y (make-array (length x)))
 (setq y (make-array '(,(length x))))
 (mvaux x y 0))

;; vector auxiliary

(defun mvaux (x y i)
 (cond ((null x) y)
      (t (setf (aref y i) (car x))
        (mvaux (cdr x) y (1+ i)))))

;; vector ?
;; already present in TI

;;(defun vectorp (x &aux y)
;;  (setq y t)
;;  (and (equal (type-of x) :array)
;;      (dotimes (z (length x) y)
;;            (setq y (and y (not (equal (type-of (aref x z))
;;                         :array)))))))
;; get i jth entry of matrix

(defun matrix-aref (m r c)
  (aref (aref m r) c))

;; set the value of the ith jth entry of a matrix

(defun matrix-setf (m r c val)
  (setf (aref (aref m r) c) val))


;; print a vector

(defun print-vector (x)
  (if (vectorp x)
      (progn
        (dotimes (y (length x) t)
                (progn (princ (aref x y)) (princ " ")))
        (terpri))))

;; matrixp
;; Xlisp code
;;(defun matrixp (x &aux y)
;;  (setq y t)
;;  (and (equal (type-of x) :array)
;;       (dotimes (z (length x) y)
;;               (setq y (and y (vectorp (aref x z)))))))

;; print a matrix

(defun print-matrix (x)
;;  (if (matrixp x)
      (terpri)
      (dotimes (y (length x) t) (print-vector (aref x y))))

;; make empty matrix

(defun make-matrix (m n &aux x)
  (setq x (make-array m))
  (dotimes (i m x) (setf (aref x i) (make-array n))))

;; make an m x n identity matrix

(defun make-Imatrix (m n &aux x)
  (setq x (make-matrix m n))
  (dotimes (i m x)
    (dotimes (j n t) (if (= i j) (matrix-setf x i j 1) (matrix-setf x i j 0)))))


;; matrix transpose operation

(defun transpose (x)
  (let* ((n (length x))                    ; # of rows
         (m (length (aref x 0)))            ; # of columns
      (r (make-matrix m n)))              ; result matrix
     (do ((i 0 (1+ i)))                  ; step through rows
        ((= i m) r)                      ; new rows complete ?
        (do ((j 0 (1+ j)))                ; step through columns
           ((= j n) t)                   ; new columns complete ?
           (matrix-setf r i j (matrix-aref x j i))))))

;; conformp returs nil if non-conforming or a list
;; (m n) representing size of result matrix

(defun conformp(a b)
 ;; (i x j) * (l x m) = (i x m) iff j = l
 (let* ((i (length a))
        (j (length (aref a 0)))
        (l (length b))
        (m (length (aref b 0))))
   (if (= j l) (list i m) nil)))

;; matrix multiplication
;; (Am x n) (Bn x p) = Cm x p
;; cij = sum(k = 1 to n)  aik * bkj

(defun m* (A B)
 (let* ((m (length A))              ;row
       (n (length (aref A 0)))       ;column A, row B
       (x (length B))               ;test conformability
       (s 0)
       (p (length (aref B 0)))       ;column B
       (C (make-matrix m p)))         ;result matrix
    (if (= n x)
     (do ((i 0 (1+ i)))
        ((= i m) C)
        (do ((j 0 (1+ j)))
           ((= j p) t)
           (matrix-setf C i j
            (progn (setq s 0)
                 (dotimes (k n s)
                        (setq s (+ s (* (matrix-aref A i k)
                                   (matrix-aref B k j))))))
           ))) 'unconformable) ))

;; chain-multiply two or more matrices
;; (m** A B C D .. N)

(defun m** (&rest x)
 (if (equal (length x) 2) (m* (car x) (cadr x)) (m**-aux x)))

(defun m**-aux (x)
 (cond ((null (cdr x)) (car x))
       (t (m* (m* (car x) (cadr x)) (m**-aux (cddr x))))))

;; Pxyz = R * Puvw   rotating --> fixed  body-attached --> reference

;; basic rotation matrices
;; the rotation matrix about the OX axis with alpha (a) angle

(defun Rxa (a)
 (setq a (radian (float a)))
 (make-vector   (make-vector 1 0 0)
            (make-vector 0 (cos (float a)) (- 0 (sin (float a))) )
            (make-vector 0 (sin (float a)) (cos (float a))) ))


;; the rotation matrix about the OY axis with phi (p) angle

(defun Ryp (p)
 (setq p (radian (float p)))
 (make-vector
           (make-vector (cos (float p)) 0 (sin (float p)))
           (make-vector  0 1 0)
           (make-vector  (- 0 (sin (float p))) 0 (cos (float p)) )))

;; the rotation matrix about the OZ axis with theta (th) angle

(defun Rzt (th)
 (setq th (radian (float th)))
 (make-vector
           (make-vector (cos (float th)) (- 0 (sin (float th))) 0 )
           (make-vector (sin (float th)) (cos (float th)) 0 )
           (make-vector 0 0 1 )))
```

APPENDIX K - THE VEHICLE CONTROLLER

"NETWORK.C
the code

```
/* network.c */

#include "/work/mcconkle/share3/shared.h"
#include "veh.h"
#include "device.h"

network(timeinterval,time,loopcnt)
float *timeinterval,*time;
long *loopcnt;
{
 extern Machine *lisp;
 extern Vehicle *vehlist,*driven;
 Vehicle *temp,*lisp_veh_search();
 char request,task;
 long veh_name;
 long vehicle_cnt();
 float delta;
 static long number=1;

#ifdef DEBUG
printf(" we are in network ");
#endif

 while (!receiver_has_data(lisp));
 read_characters(lisp,&request,1);

#ifdef DEBUG
printf(" request is %c ",request);
#endif

 while (!receiver_has_data(lisp));
  read_integer(lisp,&veh_name);

#ifdef DEBUG
printf(" veh name = %d 0,veh_name);
#endif

switch(request)
 {
 case 'T':
 case 't':
        while( !receiver_has_data(lisp));
        read_characters(lisp,&task,1);

         while( !receiver_has_data(lisp));
         read_float(lisp,&delta);
```

```
            temp = lisp_veh_search(veh_name);


            switch(task)
            {
             case 'S':
             case 's':
#ifdef DEBUG
                            printf("task s value = %f0,delta);
#endif

                           temp->vel = temp->vel + delta;
                       /* min speed is zero for now */
                           if (temp->vel < 0.0) temp->vel = 0.0;
                           /* max speed is 25 for now */
                           if (temp->vel > 24.5) temp->vel = 24.5;
                           if(temp->name == driven->name)
  setvaluator(DIAL2, (short)(driven->vel * TO_MPS * SPEEDSENS),
                      (short)((float)(MIN_SPEED*SPEEDSENS)*TO_MPS + 0.5),
                      (short)((float)(MAX_SPEED*SPEEDSENS)*TO_MPS + 0.5));
                         break;


             case 'Q':
             case 'q':
#ifdef BEBUG
                            printf("task q value = %f0,delta);
#endif

                           temp->cse = delta;

                           if(temp->cse<=0.0)
                             temp->cse = temp->cse + 360.0;
                           if (temp->cse >= 360.0)
                               temp->cse = temp->cse - 360.0;

 temp->ang = (temp->cse <= 90.0) ? DTOR*(90.0 - temp->cse)
                               : DTOR*(450.0- temp->cse);

                      if(temp->name == driven->name)
                            {
                             setvaluator(DIAL0,(int)(temp->cse * DIRSENS),
                                         (int)(-360 * DIRSENS),
                                         (int)(720 * DIRSENS));
                          }
                          break;

             case 'D':
             case 'd':
#ifdef BEBUG
                            printf("task d value = %f0,delta);
#endif

                           temp->cse = temp->cse + delta;
```

```
                    if(temp->cse<=0.0)
                      temp->cse = temp->cse + 360.0;
                    if (temp->cse >= 360.0)
                       temp->cse = temp->cse - 360.0;

  temp->ang = (temp->cse <= 90.0) ? DTOR*(90.0 - temp->cse)
                                  : DTOR*(450.0- temp->cse);

                 if(temp->name == driven->name)
                     {
                      setvaluator(DIAL0,(int)(temp->cse * DIRSENS),
                                  (int)(-360 * DIRSENS),
                                  (int)(720 * DIRSENS));
                  }
                  break;

            case 'e':
            case 'E':
#ifdef DEBUG
                      printf("task e value = %f 0,delta);
#endif
                  break;

           case 'T':
            case 't':
#ifdef DEBUG
                      printf("task t value = %f 0,delta);
#endif
                   break;

            case 'X':
            case 'x':
#ifdef DEBUG
                      printf("task x value = %f 0,delta);
#endif
                 if (delta < 0.0) delta = 0.0;
                   if (delta > 25.0) delta = 25.0;

                   temp->vel = delta;

                   if(temp->name == driven->name)

  setvaluator(DIAL2, (short)(driven->vel * SPEEDSENS),
            (short)((float)(MIN_SPEED*SPEEDSENS)*TO_MPS + 0.5),
            (short)((float)(MAX_SPEED*SPEEDSENS)*TO_MPS + 0.5));
                 break;

           default:
#ifdef DEBUG
                      printf("unrecognized task 0);
#endif
                   break;

           } /* end case task */
```

```
            while(!sender_is_free(lisp));
               write_characters(lisp,&task,1);

             send_lisp(temp);
            break;

    case 'V':
    case 'v':
#ifdef DEBUG
             printf("in vision 0);
#endif

           while(!sender_is_free(lisp));
             write_characters(lisp,&request,1);

            number = vehicle_cnt();

            while(!sender_is_free(lisp));
             write_integer(lisp,&number);

           temp = vehlist;

            while(temp!=NULL)
            {
             if(temp->name!=veh_name)
              send_lisp(temp);
            temp = temp->next;
           }

            temp = lisp_veh_search(veh_name);
            if (temp != NULL)
              send_lisp(temp);

            break;
    case 'P':
    case 'p':
#ifdef DEBUG
             printf("in perspective");
#endif

            temp  = lisp_veh_search(veh_name);
            if (temp != NULL)
               driven = temp;

           setvaluator(DIAL0,(int)(driven->cse * DIRSENS),
                          (int)(-360 * DIRSENS),
                           (int)(720 * DIRSENS));

     setvaluator(DIAL2, (short)(driven->vel * TO_MPS * SPEEDSENS),
              (short)((float)(MIN_SPEED*SPEEDSENS)*TO_MPS + 0.5),
              (short)((float)(MAX_SPEED*SPEEDSENS)*TO_MPS + 0.5));

            while(!sender_is_free(lisp));
```

```
                 write_characters(lisp,&request,1);

                send_lisp(driven);

                break;

     case 'L':
     case 'l':
#ifdef DEBUG
                 printf("in look at specific vehicle 0);
#endif

               while(!sender_is_free(lisp));
                 write_characters(lisp,&request,1);


                temp = lisp_veh_search(veh_name);

                if (temp != NULL)
                  send_lisp(temp);
                else
                  send_lisp(driven);

                break;

     case 'C':
     case 'c':
#ifdef DEBUG
                 printf("getting the time 0);
#endif

               while(!sender_is_free(lisp));
                 write_characters(lisp,&request,1);

                while(!sender_is_free(lisp));
                 write_float(lisp,&time);

                break;


     case 'U':
     case 'u':
#ifdef DEBUG
                 printf("getting the loopcount 0);
#endif

               while(!sender_is_free(lisp));
                 write_characters(lisp,&request,1);

                while(!sender_is_free(lisp));
                 write_integer(lisp,&loopcnt);

                break;
```

```
  case 'I':
  case 'i':
#ifdef DEBUG
              printf("getting the time interval 0);
#endif

          while(!sender_is_free(lisp));
            write_characters(lisp,&request,1);

           while(!sender_is_free(lisp));
            write_float(lisp,&timeinterval);

           break;
  default:  printf("unrecognized request 0);
           printf(request);
           break;

  } /* end case request */

  update_veh_pos(timeinterval);

} /* end network.c */
```

APPENDIX L - THE COMMAND AND CONTROL MODULE

METT

```
/*          Assess Tactical situations using METT                    */

/* METT is an acronym for a standard method of situation analysis that  */
/* is taught to tactical unit leaders for operational planning purposes. */

/* Using METT arrive at tactical assessments for the phases of a typical */
/* ground mission                                    */

/* Mission  -  a clear concise statement of the task to be performed.  */
/* Enemy Situation - includes composition,disposition,capabilities      */
/*              and estimated intentions.          */
/* Terrain and weather - includes key terrain, obstacle,cover and       */
/*                   concealment, fields of observation, and       */
/*                   avenues of approach.             */
/* Troops and Fire Support Available - assets available to be deployed. */

/* Results are the tasks to accomplish in 3a.Concept of Operation       */
/* in the standard Five Paragraph Order.                    */

/* pure forward chaining version. */
/* Also loads files "menu","uitility" and "forward" automatically.     */
/* To run this: Type "go(Formation,Fires_on,Atkplan)".              */

/* Setup */
setup :- consult(menu),
      consult(forward),
      consult(utility).

/* Execute the tac assessment */
go(Form,Support,Attack) :- mission,enemy,terrain_and_weather,
    troops_and_fire_support,
    listing(fact),full_forward,listing(usedfact),
     mission(Form,Support,Attack),
    my-iterate(retract(usedfact(X))).

mission(F,S,A) :- usedfact(formation(F)),
  usedfact(call_for_fire(S)),usedfact(assaultplan(A)).

done :- not(fact(Y)).
done1 :- not(usedfact(X)).
fact(default).

mission :- ask_which([
                 movement(to_contact),
               movement(to_assembly_area),
               movement(cross_open_area),
                 movement(to_cross_line_of_departure),
                    movement(to_final_coordination_line),
                    movement(to_objective),
                  assault]).
enemy :- ask_which([
    enemy_contact(unlikely),
    enemy_contact(improbable),
    enemy_contact(imminent),
    threat(front),
    threat(right),
    threat(left),
    weapon(anti-tank),
    weapon(artillary),
    weapon(machinegun),
    enemyforce(superior),
    enemyforce(inferior)
    ]).

terrain_and_weather :- ask_which([
    key_terrain,
    obstacles,
    cover_and_concealment,
    observation,
    avenue_of_approach,
    heavy_vegetation,
    night
]).

troops_and_fire_support :- ask_which([
    support(artillary),
    support(air),
    support(tanks)]).

/* Top-level task rules */
/* rule(task(' '),[pred,pred,...]). */
/* fact(task(' ')). */
/* fact(pred). */

/* movement formations */

rule(formation(column),[assaultplan(meeting_engagement)]).

rule(formation(column),[
                fire_and_manuever_flanks,
                   not(fire_and_manuever_front),
                   not(assault)]).

rule(formation(column),[speed_indicated,assaultplan(meeting_engagement)]).

rule(formation(column),[
                fire_and_manuever_flanks,
                vision_restricted,
                manuever_restricted,
                   not(assault)]).
```

```
rule(formation(wedge),[
                threat_unknown,
                dispersion_required]).

rule(formation(line),[fire_and_manuever_front,
                assault]).
rule(formation(line),[fire_and_manuever_front,
                cross_open_area]).

rule(formation(echelon_right),[fire_and_manuever_front,
                     fire_and_manuever_right,
                     vision_restricted]).

rule(formation(echelon_left),[fire_and_manuever_front,
                     fire_and_manuever_left,
                     vision_restricted]).

rule(formation(halt),[assaultplan(request_reinforcement)]).

/* type of assault to conduct (frontal or single envelopment) */

rule(assaultplan(meeting_engagement),[default,not(plan_attack)]).

rule(assaultplan(request_reinforcement),[
                      plan_attack,
                   not(fire_superiority),
                   not(cover_and_concealment),
                   not(observation),
                   not(avenue_of_approach)
                         ]).
rule(assaultplan(frontal),[
                   plan_attack,
                   fire_superiority,
                   not(observation),
                   not(cover_and_concealment),
                   not(avenue_of_approach)
                   ]).

rule(assaultplan(envelopment),[
                   plan_attack,
                   observation,
                   cover_and_concealment,
                   avenue_of_approach
                   ]).

/* fire support plan */

rule(call_for_fire(on_call),[assaultplan(meeting_engagement)]).

rule(call_for_fire(not_available),[plan_attack,not(support(artillary)),
                          not(support(air)),not(support(tanks))]).

rule(call_for_fire(not_necessary),[enemy_contact(unlikely),not(plan_attack)]).
rule(call_for_fire(key_terrain),[
                     key_terrain,
                     support(artillary),
                     not(enemy_contact(unlikely))
                    ]).
rule(call_for_fire(key_terrain),[
                     key_terrain,
                     support(air),
                     not(enemy_contact(unlikely))
                    ]).
rule(call_for_fire(objective),[plan_attack,
                   support(air),
                   not(enemy_contact(unlikely))
                  ]).

rule(call_for_fire(objective),[plan_attack,
                   support(artillary)
                  ]).
/* Definitions of intermediate predicates */
/* rule(pred,[pred,pred,....]). */

rule(plan_attack,[movement(to_cross_line_of_departure)]).
rule(plan_attack,[movement(to_final_coordination_line)]).
rule(plan_attack,[movement(to_objective)]).
rule(plan_attack,[assault]).

rule(fire_superiority,[enemyforce(inferior)]).
rule(fire_superiority,[enemyforce(superior),support(artillery)]).
rule(fire_superiority,[enemyforce(superior),support(air)]).

rule(enemyforce(inferior),[not(weapon(anti-tank)),not(weapon(artillary))]).
rule(enemyforce(superior),[weapon(anti-tank)]).

rule(speed_indicated,[movement(to_contact)]).
rule(speed_indicated,[movement(to_assembly_area)]).

rule(vision_restricted,[night]).
rule(vision_restricted,[heavy_vegetation]).
rule(vision_restricted,[obstacles]).

rule(manuever_restricted,[obstacles]).

rule(dispersion_required,[observation,
                avenue_of_approach,
                not(enemy_contact(unlikely)),
                not(cover_and_concealment),
                not(obstacles),
                not(vision_restricted)]).


rule(threat_unknown,[not(threat(front)),not(threat(left)),not(threat(right))]).

rule(fire_and_manuever_front,[threat(front)]).
rule(fire_and_manuever_left,[threat(left)]).
rule(fire_and_manuever_right,[threat(right)]).
rule(fire_and_manuever_flanks,[fire_and_manuever_left,fire_and_manuever_right]).

/* Question decoding */
/* questioncode(pred,'question about pred - choose if true'). */
/* questioncode(pred_with_var(X),X) :- write('did pred_with_var X occur '). */

questioncode(movement(X),X) :- write('The unit is moving ').
questioncode(assault,'The unit is attacking an objective').

questioncode(enemy_contact(X),X) :- write('Enemy contact is ').
questioncode(threat(X),X) :- write('Direction of threat is from ').
questioncode(enemyforce(X),X) :- write('Enemy force in relation to unit is ').
questioncode(weapon(X),X) :- write('Enemy employs ').

questioncode(cntrlpt(X),X) :- write('following control point established').
questioncode(key_terrain,'Key terrain about the objective ').
questioncode(obstacles,'axis of movement contains obstacles or hills').
questioncode(cover_and_concealment,'Concealment exists for attack element ').
questioncode(observation,'base of fire has good observation of the objective ').
questioncode(avenue_of_approach,'Approach to the objective on its flanks').

questioncode(heavy_vegetation,'terrain has heavy vegetation').
questioncode(night,'Night movement').

questioncode(support(X),X) :- write('Unit has direct support of ').

:- setup.
```

APPENDIX M - THE DISPLAY LOOP CONTROLLER

EVENT.C

```
/*****************************************************************************
* FILENAME : event.c                                        *
* CONTAINS : event()                                        *
* CALLED BY: main()                                         *
* CALLS   : readcontrols(), update_veh_pos(), update_view_pos(),  *
*         update_look_pos(), view_bounds(), edit_navbox(), edit_indbox(),  *
*         display_terrain(), handleMENU(), handleMAP(), update_vrh_grid()  *
* LAST MODIFIED: 11/02/87                                   *
*****************************************************************************/

#include "gl.h"
#include "device.h"
#include <sys/types.h>
#include <sys/times.h>
#include "veh.h"
#include "stdio.h"
#include "/work/mcconkle/share3/shared.h"

event(unmask,greys,tank,jeep,truck,junk,intank,vehicon,scrn,
    menu,text,inst,speedometer,maxitems,mtag)

Colorindex unmask;
Boolean   *greys;
Object    tank,jeep,truck,junk,intank,vehicon[];
Object    scrn[],menu[],text[],inst[];
Object    speedometer;
short     maxitems[];
Tag       mtag[][NUMMENUITEMS];
{
 extern Machine   *lisp,SYM1,SYM3,SYM4,TI;
 extern Gridnode *vehgrid[NUMZGRIDS][NUMXGRIDS];
 extern Vehicle  *vehdata[NUMVEHTYPES][MAXVEH], *vehlist ,*driven;
 extern Coord    gridcoord[NUMZGRIDS][NUMXGRIDS][2][3][3],gndplane[4][3];
 extern long     gridcolor[NUMZGRIDS][NUMXGRIDS],gndplane_color,contourcolor;
 extern Boolean  stahl;
 Boolean    act=TRUE,sel=FALSE,vehiclehit=FALSE;
 Boolean    defaults_set=FALSE;
 Boolean    zoomed=FALSE,array_built=FALSE;
 struct tms  timeinfo;
 Object     navbox,indbox;
 Device     dev,sx,sy,val,lsx,M3stat;
 float      framerate=0.0;
 short      fov,hittype,hitindex,stat= DECIDING;
 short      wy,lwy,item,litem,vehtype,numveh[NUMVEHTYPES];
 short      i,mcnt,scnt,insocket,outsocket;
 float      tilt,lookang,lookdeg;
 static float lastsec;
static float elapsedsec;
Coord      glx,glz,grx,grz,windowsx,windowsy;
Coord      px,py,pz;
long       loopcnt=0,start,elapsed60ths;
Tag        arrowtag,turnvehtag,headingtag,speedtag,zoomtag,positiontag;

for(i=0;i<NUMVEHTYPES;i++) numveh[i]=0;
lwy=311; item=litem=(-1); sx=lsx=895; windowsx=windowsy=0.0;
mcnt=OPENING; scnt=INTRO1;
setcursor(0,GREEN,unmask);
setvaluator(MOUSEY,710,400,767);
setvaluator(MOUSEX,895,783,1008);
attachcursor(MOUSEX,MOUSEY);

qdevice(MOUSE3);qdevice(MOUSEY);
tie(MOUSE3,MOUSEX,MOUSEY);
frontbuffer(TRUE);
callobj(scrn[INTRO1]);
callobj(menu[OPENING]);
frontbuffer(FALSE);
qreset();
curson();

while(act) {
 if(stat==DRIVING) {
  loopcnt++;
  lastsec=times(&timeinfo);
  elapsed60ths=lastsec-start;
  elapsedsec=(float)(elapsed60ths)/60.0;

  if(elapsedsec!=0.0) framerate += 1/elapsedsec;

  if(loopcnt%100 == 0) {
  printf("avg frame rate: %f0,framerate/loopcnt);
  printf("frame rate this frame: %f0,1/elapsedsec);
  }

  start=lastsec;

  if(receiver_has_data(&TI))
   {
   lisp = &TI;
   network(elapsedsec,lastsec);
   }

  if(receiver_has_data(&SYM3))
   {
   lisp = &SYM3;
   network(elapsedsec,lastsec);
   }

  if(receiver_has_data(&SYM1))
   {
   lisp = &SYM1;
  network(elapsedsec,lastsec);
  }

 if(receiver_has_data(&SYM4))
  {
  lisp = &SYM4;
  network(elapsedsec,lastsec);
  }

 read_controls   (&*greys,&act,&lookang,&tilt,&lookdeg,&fov);

 update_veh_pos  (elapsedsec);

 view_bounds     (lookang,fov,&glx,&glz,&grx,&grz);

 update_veh_grid (lookang);

 update_look_pos (lookang,tilt,&px,&py,&pz);

 edit_navbox     (navbox,lookang,fov,arrowtag,positiontag);

 edit_indbox     (indbox,lookdeg,fov,speedtag,headingtag,turnvehtag,zoomtag);

 display_terrain (lookang,glx,glz,grx,grz,px,py,pz,fov,
             tank,jeep,truck,intank);

 writemask(SAVEMAP);
 callobj(navbox);
 writemask(unmask);
 callobj(indbox);
 swapbuffers();
 }


if(vehiclehit)   casualty(vehiclehit,zoomed,scrn,menu,vehicon,windowsx,
                  windowsy,&mcnt,&stat,hittype,hitindex);

while(qtest() != 0) {
 dev=qread(&val);
 if(dev==MOUSEX) {
   lsx=sx;
   sx=val;
 }
 else if(dev==MOUSEY) {
   sy=val;
   wy=sy-399;
   item=5-(wy/50);
 }
 else if(dev==MOUSE3) {
   M3stat=val;
   lsx=sx;
   qread(&sx);
   qread(&sy);
 }
```

```
unqdevice(MOUSE3);unqdevice(MOUSEY);tie(MOUSE3,0,0);
if(mcnt>=3) unqdevice(MOUSEX);

if(sx>=783)
handlemenu(unmask,&defaults_set,&act,&sel,
        &zoomed,&array_built,
        dev,sx,M3stat,lsx,scrn,menu,text,
        inst,vehicon,&navbox,&indbox,numveh,item,&stat,
        &insocket,&outsocket,wy,&lwy,&litem,maxitems,&mcnt,
        &scnt,&vehtype,&windowsx,&windowsy,&headingtag,&turnvehtag,
        &arrowtag,&speedtag,&zoomtag,&positiontag
        ,mtag);

else if(sx<768)
handlemap(unmask,greys,&zoomed,&array_built,
        sx,sy,dev,lsx.M3stat,
        &indbox,&navbox,speedometer,vehicon,menu,scrn,inst,text,&tilt,
        &windowsx,&windowsy,&lookdeg,&stat,&mcnt,&fov,numveh,vehtype,
        &litem,maxitems,&start,mtag);

sx=getvaluator(MOUSEX);
qdevice(MOUSE3);qdevice(MOUSEY);tie(MOUSE3,MOUSEX,MOUSEY);
if(mcnt>=3) qdevice(MOUSEX);
if(mcnt<4) qreset();
}
}
}
```

APPENDIX N - IRIS TO LISP MACHINE COMMUNICATION

SENDLISP.C

```
#include "veh.h"
#include "/work/mcconkle/share3/shared.h"
#include "device.h"
#include "gl.h"
#include "stdio.h"
send_lisp(vname)
Vehicle *vname;
{
 extern Machine lisp;

#ifdef DEBUG
    printf("made it to send_lisp0);
    printf(" vname->name %d0,vname->name);
    printf(" vname->x    %f0,vname->x);
    printf(" vname->y    %f0,vname->y);
    printf("        z    %f0,vname->z);
    printf("        vel  %f0,vname->vel);
    printf("        cse  %f0,vname->cse);
#endif

/* send the lisp machine a complete object */

    while(!sender_is_free(&lisp));
    write_integer(&lisp,&vname->name);
    while(!sender_is_free(&lisp));
    write_float(&lisp,&vname->x);
    while(!sender_is_free(&lisp));
    write_float(&lisp,&vname->y);
    while(!sender_is_free(&lisp));
    write_float(&lisp,&vname->z);
    while(!sender_is_free(&lisp));
    write_float(&lisp,&vname->vel);
    while(!sender_is_free(&lisp));
    write_float(&lisp,&vname->cse);

}
```

APPENDIX O - DYNAMIC SPEED CHANGE

CHANGESPEED.C

```
#include "gl.h"
#include "veh.h"
#include "device.h"

changespeed()
 {
  extern Vehicle *driven;
  float deltat = 0.15;  /* elapsed time between frames */
  float speedgain = 0.2;


   driven->spd = (float)(getvaluator(DIAL2) /( SPEEDSENS ));
   driven->vel = driven->vel + speedgain *
   (driven->spd - driven->vel)* deltat;
   if((driven->vel > -1)&&(driven->vel < 1)&&(driven->spd == 0))
     driven->vel = 0;
 }
```

APPENDIX P - DYNAMIC COURSE CHANGE

CHANGECSE.C

```
#include "gl.h"
#include "veh.h"
#include "device.h"


/* change course based on the steering angle - vehicle
  dynamics are incorporated into this function */


#include "gl.h"
#include "veh.h"
#include "device.h"

changecse(lastcsedeg,deltacsedeg)
float *deltacsedeg;
float *lastcsedeg;
{
 float deltat,tankturn,jeepturn;
 extern Vehicle *driven;
 deltat = 0.15;    /* elapsed time between frames */
 tankturn = 0.2;   /* turn gain for tanks */
 jeepturn = 0.3;   /* turn gain for jeeps */

 driven->ster = (float)getvaluator(DIAL3) / DIRSENS;
  if ((driven->ster > -1.0)&&(driven->ster < 1.0))
    driven->ster = 0.0;

 if(driven->ster != 0) {
 /* if steering angle > 0 */
  if(driven->t == TANKS) {
   /* tanks can turn when speed is 0 */
    driven->cse = *lastcsedeg + tankturn * driven->ster * deltat;
    if(driven->cse >= 360.0)
       driven->cse -= 360.0;
    else if(driven->cse < 0.0)  {
       driven->cse += 360.0;
     }
    setvaluator(DIAL0,(int)(driven->cse*DIRSENS),
             (int)(-360*DIRSENS), (int)(720*DIRSENS));
    *deltacsedeg = driven->cse - *lastcsedeg;
   }
  else if(driven->vel != 0.0) {
    /* jeeps can turn only if speed > 0 */
    driven->cse = *lastcsedeg + jeepturn * driven->ster *
                 driven->vel * deltat;
    if(driven->cse >= 360.0)
       driven->cse -= 360.0;
    else if(driven->cse < 0.0) {
      driven->cse += 360.0;
    }
    setvaluator(DIAL0,(int)(driven->cse*DIRSENS),
            (int)(-360*DIRSENS), (int)(720*DIRSENS));
    *deltacsedeg = driven->cse - *lastcsedeg;
  }
  else *deltacsedeg = 0.0;


 }
}
```

APPENDIX Q - NON-DYNAMIC COURSE CHANGE

NEWCOURSE.C

```
/* absolute course change - no vehicle dynamics */

#include "gl.h"
#include "veh.h"
#include "device.h"

newcourse(lastcsedeg,deltacsedeg)
float *lastcsedeg,*deltacsedeg;
{
 extern Vehicle *driven;

  if((int)driven->ster == 0) {
   /* if the steering angle is 0 the non-dynamic turning is enabled */
   if(driven->t == TANKS) {
    /* tanks can turn when speed = 0 */
    driven->cse = (float)getvaluator(DIAL0) / DIRSENS;
    *deltacsedeg= driven->cse - *lastcsedeg;
    if (driven->cse >= 360.0) {
      driven->cse -= 360.0; setvaluator(DIAL0,(int)(driven->cse*DIRSENS),
               (int)(-360*DIRSENS), (int)(720*DIRSENS));
    }
    else if (driven->cse < 0.0) {
      driven->cse += 360.0; setvaluator(DIAL0,(int)(driven->cse*DIRSENS),
               (int)(-360*DIRSENS), (int)(720*DIRSENS));
    }
   } /* end if driven->t == TANKS */
   else {
    /* jeeps */
    if( driven->vel != 0) {
    /* jeeps can only turn when speed > 0 */
      driven->cse = (float)getvaluator(DIAL0) / DIRSENS;
      *deltacsedeg= driven->cse - *lastcsedeg;
      if (driven->cse >= 360.0) {
        driven->cse -= 360.0; setvaluator(DIAL0,(int)(driven->cse*DIRSENS),
               (int)(-360*DIRSENS), (int)(720*DIRSENS));
      }
     else if (driven->cse < 0.0) {
     driven->cse += 360.0; setvaluator(DIAL0,(int)(driven->cse*DIRSENS),
               (int)(-360*DIRSENS), (int)(720*DIRSENS));
      }
     }
    else *deltacsedeg = 0.0;
    }
   } /* end else */
}
```

APPENDIX R - AUGMENTED TRANSITION NETWORK PARSER

ATN

```
;;;;;;;;;;;;;;;;;;;;;;  globals ;;;;;;;;;;;;;;;;;;;;;;;;;

(setq debug       nil      ;flag for debug functions in the atn to exec
    *dictionary* nil       ;list of all words known by atn
    *features*  nil        ;all features (morphology) of those words
    *agenda*    nil
    *actions*   nil
    *sentence*  nil
    *uselist*   nil;save-last and use
    *q*         nil
    *token*     nil
    *sem-end*   nil        ;all semantics functions on the end queue
    *sem-wait*  nil        ;all semantics functions on the wait queue
    *stk*       nil)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; uncommon lisp stuff for xlisp       ;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun debugon()
 (progn (setq debug t) (break "debug mode on") t))

(defun cc () (continue))

(defun display (&optional fp)
  (progn
   (if (null fp) (setq fp *standard-output*))
   (print '------actions------ fp)
   (print '------agenda------- fp) (print (car *agenda*) fp)
   (print '------*actions*---- fp) (print *actions* fp)
   (print '------queue-------- fp) (print *q* fp)
   (print '------sentence----- fp) (print *sentence* fp)
   (print '------stack-------- fp) (print *stk* fp)
   (print '------token-------- fp) (print *token* fp)))

(defun snapshot (x)
 (let ((fp (openo (string-append "snapshot." (symbol-name x)))))
    (progn (display fp) (close fp) t)))

;; no loop in xlisp

(DEFMACRO LOOP
      (A &optional B C D E F G H I J K L M)
      (BACKQUOTE (PROG NIL
                 TAG
                 (COMMA A)
```

```
                  (COMMA B)
                  (COMMA C)
                  (COMMA D)
                  (COMMA E)
                  (COMMA F)
                  (comma G)
                  (comma h)
                     (COMMA I)
                     (COMMA J)
                     (COMMA K)
                     (COMMA L)
                     (COMMA M)
                  (GO TAG) )) )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; uncommon lisp stuff for xlisp      ;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; common lisp stuff for xlisp        ;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(setq string-append #'strcat)

(defun implode (x)
 (make-symbol (lst->str x)))

(DEFUN lst->str (X)
  (COND ((NULL (CDR X)) (SYMBOL-NAME (CAR X)))
        (T (STRCAT (SYMBOL-NAME (CAR X)) (lst->str (CDR X))))))

(defun explode (x)
 (str->lst (symbol-name x)))

(DEFUN str->lst (X)
 (DO* ((POS 1 (1+ POS))
      (LNG 1)
      (STRNG (SUBSTR X POS LNG) (SUBSTR X POS LNG))
      (SYM (MAKE-SYMBOL STRNG) (MAKE-SYMBOL STRNG))
      (LST (LIST SYM) (CONS SYM LST)))
      ((= POS (LENGTH X)) (reverse lst))))

;;;;;;; end xlisp specific code ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; symbol primitives needed to break up and find roots etc ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun newsym (sym)
 (intern (symbol-name (gensym (symbol-name sym)))))

(defun make$ (sym)
 (intern (string-append "$" (symbol-name sym))))
```

```
(DEFUN SYM-EQ (X Y)
 (EQUAL (SYMBOL-NAME X) (SYMBOL-NAME Y)))

(defun sym-eql (x y)
 (cond  ((or (not (atom x)) (not (atom y))) nil)
        ((null x) (and (null x) (null y)))             ;if null x then null y
        ((null y) nil)                                 ;if null y (not null x)
        (t (sym-eql-aux (explode x) (explode y)))))     ;must be symbols

(defun sym-eql-aux (x y)
 (cond ((null x) t)
       ((null y) nil)
       (t (and (sym-eq (car x) (car y)) (sym-eql-aux (cdr x) (cdr y))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;  tree manipulation primitives and such ;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; returns first occurance of node in tree l

(defun search (x l)
 (cond ((null l) nil)
       ((atom (car l)) (if (sym-eql x  (car l))
                          l (search x (cdr l))))
       (t (or (search x (car l)) (search x (cdr l))))))

;;; returns last occurance of node in tree l

(defun getlast (x l &optional answer)
 (cond ((null l) answer)
       ((atom l) (if (sym-eql x l) l answer))
       ((atom (car l)) (if (sym-eql x (car l))
                          (progn (setq answer l) (getlast x (cdr l) answer))
                          (getlast x (cdr l) answer)))
       (t (or (getlast x (cdr l)answer) (getlast x (car l)answer)))))

;;; prints the tree

(DEFUN PRTTREE (X &optional SPACES fp)
 (IF (NULL SPACES) (SETQ SPACES 1) (SETQ SPACES (+ SPACES 5)))
 (if (null fp) (setq fp *standard-output*))
 (COND ((NULL X) T)
       ((and (equal (length x) 2)
           (atom (cadr x)))
             (dotimes (y spaces t) (princ " " fp))
             (princ (car x) fp) (princ " -- " fp) (print (cadr x) fp) )
      ((ATOM (CAR X))
       (DOTIMES (Y SPACES T) (PRINC " " fp))
       (PRINT (CAR X) fp)
       (DOLIST (Z (CDR X) T) (PRTTREE Z SPACES fp)))
      (T (DOLIST (Z X T) (PRTTREE Z SPACES fp)))))


;;; return name of branch of tree
```

```
(defun nodename(x) (car x))

;;; return nodes branches

(defun branches (x)
 (cdr x))

;;; pred to determine if this node the chosen one

(defun thenode (name tree)
 (cond ((nodep tree)
        (sym-eql name (nodename tree)))
       (t nil)))


;;; is this node a leaf ?

(defun leafp (node)
 (and (not (atom node))
      (atom (car node))
      (equal (length node) 2) (atom (cadr node))))

;;; is this structure a node ?

(defun nodep (node &aux bool)
 (setq bool t)
 (or (leafp node)
   (and (not (atom node)) (atom (car node)) (>= (length node) 2)
      (dolist (b (branches node) bool)
            (setq bool (and bool (nodep b)))))))

;;; delete a branch and all of its sub branches from the tree

(defun prune (name x &aux good temp)
 (setq good nil)
 (cond ((atom x) x)
      ((thenode name x) nil)
      ((nodep x)
      (dolist (branch (branches x) (cons (nodename x) (reverse good)))
            (if (setq temp (prune name branch))
                 (setq  good (cons temp good))
                 good)))
      (t x)))

;;; graft on a branch to the tree

(defun graft (name y x &aux good temp)
 (setq good nil)
 (cond ((atom x) x)
      ((thenode name x) (cons (nodename x) (cons y (branches x))))
      ((nodep x)
      (dolist (branch (branches x) (cons (nodename x) (reverse good)))
            (if (setq temp (graft name y branch))
                 (setq  good (cons temp good))
```

```
                good)))
        (t x)))

;;; replace a branch on the tree with another branch

(defun replace(name y x &aux good temp)
 (setq good nil)
 (cond ((atom x) x)
       ((thenode name x)
       (cond ((listp y) (cons (nodename x) y))
             (t (list (nodename x) y))))
       ((nodep x)
       (dolist (branch (branches x) (cons (nodename x) (reverse good)))
             (if (setq temp (replace name y branch))
                 (setq  good (cons temp good))
                 good)))
       (t x)))

;;; strips non-terminals from parse tree and
;;; returns a tree of terminal symbols only

(defun reducetree (x &aux good temp)
 (setq good nil)
 (cond ((leafp x) (cadr x))
       ((nodep x)
       (dolist (branch (branches x) (reverse good))
             (if (setq temp (reducetree branch))
                 (setq  good (cons temp good))
                 good)))
       (t x)))

;;; strips non-terminals from parse tree and
;;; returns a tree of the last non-terminal symbols only

(defun analyzetree (x &aux good temp)
 (setq good nil)
 (cond ((leafp x) (car x))
       ((nodep x)
       (dolist (branch (branches x) (reverse good))
             (if (setq temp (analyzetree branch))
                 (setq  good (cons temp good))
                 good)))
       (t x)))

;;; flattens a tree to a single list of symbols

(defun squash (s)
 (cond ((null s) nil)
       ((atom s) (list s))
       (t (append (squash (car s)) (squash (cdr s))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;; end tree manipulation primitives and such ;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;; the dictionarv facility ;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; dictionary entry has a property list:
;;;     root-form
;;;     part-of-speech
;;;     feature-assignments .... association list ((tense past) (number 1))
;;;             tense = past or present or future or tenseless
;;;          number= 1s 2s 3s 1p 2p 3p
;;;


(defmacro addword (word root-form part-of-speech &rest feature-assignments)
 '(progn
   (setf (get ',word 'lex) ',word)
   (setf (get ',word 'root) ',root-form)
   (setf (get ',word 'type) ',part-of-speech)
   (setf (get ',word 'features) ',feature-assignments)
   (setq *dictionary* (cons ',word *dictionary*))
   t))

(defmacro dictionary (word part root &rest x)
 '(setf (get ',word 'features) ',x))

;;;     define dimensions  default values  other values

        (defmacro def-feature (dim default &rest other-values)
         '(setq *features*
              (cons (list ',dim ',default ',other-values) *features*)))

        (DEFMACRO FEATURE (X)
         (BACKQUOTE (ASSOC (QUOTE (COMMA X)) *FEATURES*)))

        (DEFMACRO DEFAULT (X)
         (BACKQUOTE (CADR (FEATURE (COMMA X)))))

        (DEFMACRO OTHER (X)
         (BACKQUOTE (CADDR (FEATURE (COMMA X)))))

        (defmacro featurep (x)
         '(not (null (feature ,x))))

        (defmacro feature-value-p (f v)
         '(or  (member ,v  (default ,f) :test #'equal)
              (member ,v  (other ,f)   :test #'equal)
              (memb̈er 'anything (other ,f) :test #'equal)))


(defmacro d-the (dimension of constituent &aux temp)
 '(progn
  (if (sym-eql '$  ',constituent)
       (setq temp
           (cadr (assoc ',dimension (get ,constituent 'features))))
```

```
    (setq temp
          (cadr (assoc ',dimension (get ',constituent 'features)))))
  (if (null temp) (setq temp (default ',dimension)))
  (if (equal temp 'anything) (setq temp nil))
  temp))


;;; differs from the book on ':='
;;;     (:= mood of $s-maj  'question)
;;; instead of
;;;     (:= (the mood of $s-maj) 'question)

(defmacro := (dimension of constituent value)
 '(if (sym-eql '$ ',constituent)

      (if    (feature-value-p ,dimension ,value)
             (setf  (get ,constituent 'features)
                    (cons         (list ',dimension ,value)
                           (get ,constituent 'features)))
             '"unknown feature or feature value")

      (if    (feature-value-p ,dimension ,value)
             (setf  (get ',constituent 'features)
                    (cons (list ',dimension ,value)
                           (get ',constituent 'features)))
             '"unknown feature or feature value")
))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;; Parser primitives used in atn-main of the interpreter ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;     queue mgmt to build the tree

(DEFUN QUEUE (X L)
 (INVERT (CONS X (INVERT L))))


(DEFUN INVERT (L)
 (COND ((NULL L) NIL)
       (T (CONS (LAST L) (INVERT (BUTLAST L))))))


(DEFUN BUTLAST (L)
 (COND ((NULL (CDR L)) NIL)
       (T (CONS (CAR L) (BUTLAST (CDR L))))))


(DEFUN LAST (L)
 (COND ((NULL (CDR L)) (CAR L))
       (T (LAST (CDR L)))))

(defmacro mapq1 (func parm q &aux temp-mpq1)
 '(cond ((null ,q) nil)
        (t (setq temp-mpq1 (,func ,parm (car ,q)))
        (if (null temp-mpq1) (mapq1 ,func ,parm (cdr ,q))
            (cons temp-mpq1 (mapq1 ,func ,parm (cdr ,q)))))))

(defmacro mapq2 (func parm1 parm2 q &aux temp-mpq2)
 '(cond ((null ,q) nil)
       (t (setq temp-mpq2 (,func ,parm1 ,parm2 (car ,q)))
        (if (null temp-mpq2) (mapq2 ,func ,parm1 ,parm2 (cdr ,q))
            (cons temp-mpq2 (mapq2 ,func ,parm1 ,parm2 (cdr ,q)))))))

;;;    stack functions to build the tree

(defun push (x)
 (setq *stk* (cons x *stk*)))

(defun topstk ()
 (car *stk*))

(defun pop(&aux ret)
 (setq  ret  (car *stk*))
 (setq *stk* (cdr *stk*))
 ret)

(defun pushtoken (x)
 (setq *token* (cons x *token*)))

(defun toptoken ()
 (car *token*))

(defun poptoken(&aux ret)
 (setq  ret  (car *token*))
 (setq *token* (cdr *token*))
 ret)

;;;    the advertized stuff in chapter 4
;;;

;;;    look at input stream, if next symbol terminal-symbol
;;;    remove it from input stream and return true

       (defmacro category (term)
        '(progn
          (setq $last (car *sentence*))
         (if (equal (get (car *sentence*) 'type) ',term)
              (progn
                  (setq *q*
                    (queue (list ',term (car *sentence*)) *q*))
                  (setq *sentence* (cdr *sentence*))
                 t))))

;;;    give networks names and define grammers

       (defmacro def-net (net-name network)
       '(progn (setq ,net-name ',network) ',net-name))
```

```
;;;   detach constituent location

      (defun detach (c l &aux templ)
       (setq templ (last (mapq1 getlast l *q*)))
      (if (eval debug) (break "detach"))
       (if (null templ)
          (progn
            (setq templ (mapq1 getlast l *stk*))
            (if (eval debug) (break "detach"))
            (if (null templ)
               nil
               (progn
                   (setq templ (prune c templ))
                   (setq *stk* (mapq2 replace l (cdr templ) *stk*))
                   (if (eval debug) (break "detach")) t
                   )))
          (progn
            (setq templ (prune c templ))
            (setq *q* (mapq2 replace l (cdr templ) *q*))
            (if (eval debug) (break "detach")) t
            )))

;;;   drop a piece of the tree structure back into the input stream

      (defun drop (x &aux temp)
       (progn
            (setq temp (last (mapq1 getlast x *q*)))
            (if (eval debug)(break "drop1: temp"))
            (if (null temp) nil
               (progn
                   (if (eval debug)(break "drop1-in: sentence"))
                   (setq *sentence* (append (squash (reducetree temp))
                                          *sentence*))
                   (setq *q* (mapq1 prune x *q*))
                   t ))

            (if (null temp)
               (progn
                   (setq temp (last (mapq1 getlast x *stk*)))
                   (if (eval debug)(break "drop2: temp"))
                   (if (null temp) nil
                      (progn
                      (if (eval debug)(break "drop2-in: sentence"))
                      (setq *sentence* (append  (squash (reducetree temp))
                                             *sentence*))
                      (setq *stk* (mapq1 prune x *stk*))
                      (if (search x *stk*)
                          (setq *stk* (cons (mapq1 prune x (car *stk*))
                                         (cdr *stk*))))
                       t ))))

                   (if (null temp) nil t)))

;;;   translate a piece of the tree structure back into the structure stuff
```

```
     (defun analyze (x &aux temp)
      (progn
           (setq temp (last (mapq1 getlast x *q*)))
           (if (null temp)
              (setq temp (last (mapq1 getlast x *stk*))))
           (squash (analyzetree temp))))

;;;  assign right side value to left side
;;;  (defmacro := (L R)
;;;   (setf ,l ,R))

;;;  (== "left-hand-side" right-hand-side)

     (defun == (x y) (sym-eql x y))

;;;  constituent is inserted into the input stream

     (defmacro insert (c)
      '(if   (listp ',c)
           (setq *sentence* (cons ,c *sentence*))
           (setq *sentence* (cons ',c *sentence*))))

;;;  look at input stream, if next symbol terminal-symbol
;;;  return true

     (defmacro peek (term)
      '(progn
           (setq $last (car *sentence*))
           (equal (get (car *sentence*) 'type) ',term)))

;;;  puts last parsed object into a special list which the use command
;;;  knows about

     (defun save-last ()
      (setq *uselist* (cons $last *uselist*)))

;;;  returns the first (leftmost in tree) node of type category that
;;;  is found directly under constituent

     (defmacro the-first (category of constituent &optional key &aux answer)
      '(if (equal ',category 'nil) nil
        (progn
          (setq answer nil)
          (if (d-the ,category of ,constituent)
             (setq answer (d-the ,category of ,constituent)))

          (if (and (null answer)
                  (or (sym-eql (car *token*)  ',constituent)
                     (sym-eql (car *token*)  ,constituent)))
             (setq answer (car (mapq1 search ',category *q*))))

          (if (null answer)
          (if (or (sym-eql '$  ',constituent)
                 (listp ',constituent))
          (setq answer (car
           (or
           (mapq1 search ',category (mapq1 search ,constituent *stk*))
           (mapq1 search ',category (mapq1 search ,constituent *q*)))))
          (setq answer (car
           (or
           (mapq1 search  ',category (mapq1 search ',constituent *stk*))
          (mapq1 search  ',category (mapq1 search ',constituent *q*))))))

          (cond      ((atom answer) answer)
                ((leafp  answer)
                 (if (null ,key) (cadr answer) (nodename answer)))
                ((nodep answer)
                 (if (null ,key) (branches answer) (nodename answer)))
                (t answer))
)))

;;;   returns the last (rightmost in tree) node of type category that
;;;   is found directly under constituent

;;;    (last (mapq1 getlast x *stk*))

      (defmacro the (category of &optional constituent key &aux answer)
      '(if (equal ',category  'nil) nil
       (if (equal ',category 'word) ',of
          (progn
          (setq answer nil)
          (if (d-the ,category of ,constituent)
             (setq answer (d-the ,category of ,constituent)))

          (if (eval debug) (break "the: answer"))

          (if (and (null answer)
                 (or (sym-eql (car *token*)  ',constituent)
                    (sym-eql (car *token*)   ,constituent)))
            (setq answer (last (mapq1 getlast ',category *q*))))

          (if (null answer)
          (if (or (sym-eql '$  ',constituent)
                 (listp ',constituent))
          (setq answer (last
           (or
      (mapq1 getlast ',category (mapq1 getlast ,constituent *q*))
      (mapq1 getlast ',category (mapq1 getlast ,constituent *stk*)))))
          (setq answer (last
           (or
      (mapq1 getlast  ',category (mapq1 getlast ',constituent *q*))
      (mapq1 getlast  ',category (mapq1 getlast ',constituent *stk*)))))))

          (if (eval debug) (break "the end"))

          (cond      ((atom answer) answer)
                ((leafp answer)
                 (if (null ,key) (cadr answer) (nodename answer)))
```

```
                    ((nodep answer)
                     (if (null ,key) (branches answer) (nodename answer)))
                    (t answer))
                    ))))

;;;     inserts the last item saved into the input stream
;;;

        (defun use(&aux ret)
          (setq ret (car *uselist*))
          (if (not (null ret))
              (progn
                    (setq *uselist* (cdr *uselist*))
                    (setq *sentence* (cons ret *sentence*)))))

;;;     look at input stream, if next word
;;;     from input stream has root remove it and return true

        (defmacro word (root)
         '(progn
              (setq $last (car *sentence*))
              (if (equal (get (car *sentence*) 'root) ',root)
              (progn
                    (setq *q*
                      (queue (list ',root (car *sentence*)) *q*))
                    (setq *sentence* (cdr *sentence*))
                  t))))



;;;    backtracking atn interpreter


;;;; agenda       = (choice-point1 choice-point2 ...)
;;;; choice-point = (sentence actions)
;;;; sentence     = '(some words like this in a list)
;;;; actions      = '(list of actions to perform)

(defun store_choicepoint (s actions queue stk token)
  (setq *agenda* (cons (list s actions queue stk token) *agenda*)))

(defun backto_choicepoint ()
  (setq *sentence* (caar *agenda*))
  (setq *actions* (list (cadar *agenda*)))
  (setq *q*    (caddar *agenda*))
  (setq *stk*    (car (cdddar *agenda*)))
  (setq *token*   (cadr(cdddar *agenda*)))
  (setq *agenda* (cdr *agenda*)))


;;; function atn

;;; args: *s*, the input sentence - a nonlocal variable.
;;; vars: *agenda* - stores the choice-points put on by
;;;               the either command.  Initially it is
;;;               set to the (parse s-maj) action, and *s* to
;;;               the input sentence.
;;; algorithm:
;;; loop: Until either a successful parse, or *agenda* is empty.
;;;       Call atn-main choosing the first choice-point from
;;;       *agenda*, and removing it from the list.  If successful,
;;;       print out resulting tree; else try next choice-point.
;;; end loop.

(defun atn (sentence start-action)
 (setq *sentence* sentence)
 (setq *agenda*  '((nil (done atn) nil nil nil)) )
 (setq *actions* nil)
 (setq *q* nil)
 (store_choicepoint sentence start-action nil nil nil)
 (catch 'done
      (loop
        (if (eval debug) (break "atn start"))
       (backto_choicepoint)
       (if (null *actions*) (throw 'done *sentence*))
       (if (atn-main *actions*) (throw 'done 't)))))

;;; function atn-main
;;;
;;; args: actions - A list of actions to be performed.
;;;
;;; loop: Until either no more actions on actions or an action fails.
;;;       Remove first action from actions and eval it.
;;;
;;;       case: category, word, peek, = etc: If test
;;;            is successful, then continue.
;;;            otherwise report failure to atn.
;;;
;;;            seq: Put all of the subactions on front of actions.
;;;
;;;            either: pick one of the possibilities "at random"
;;;                   and put it on the front of actions.
;;;                     For each alt action, store a choice-point
;;;                     with (a) current *s*
;;;                         (b) the alt action added to actions.
;;;
;;;            parse: Add a done action to actions.  Put the network
;;;                  associated with the constituent to be parsed on
;;;                  actions.
;;;
;;;            done: The parser has completed a constituent.  If there
;;;                 are not further actions, then, if *s* is empty,
;;;                 report back success, and if *s* has things left on
;;;                 it, report back failure.

(defun atn-main (start &aux actions node temp)
 (progn
   (setq actions start)
```

```
(catch 'exit
   (loop
        (if (eval debug) (break "atn-main"))
        (cond ((null actions)
               (if (null *sentence*)
                     (throw 'exit t)
                 (throw 'exit nil)))

             ((listp (caar actions)) (setq actions (car actions)))

             ((or
               (equal (caar actions) 'category)
               (equal (caar actions) 'word)
               (equal (caar actions) 'peek)
               (equal (caar actions) '==)
               (equal (caar actions) ':=)
               (equal (caar actions) 'insert)
               (equal (caar actions) 'debugon)    ;turn debug mode on
               (equal (caar actions) 'drop)
               (equal (caar actions) 'detach)
               (equal (caar actions) 'the)
               (equal (caar actions) 'the-first)
               (equal (caar actions) 'save-last)
               (equal (caar actions) 'semantics)
               (equal (caar actions) 'snapshot)  ;snap shot atn
               (equal (caar actions) 'use))
              (if (eval (car actions))
                 (setq actions (cdr actions))
                 (throw 'exit nil)))

             ((equal (caar actions) 'optional)
             (if (null *sentence*)
                  (setq actions (cdr actions))
                  (progn
                      (store_choicepoint
                                   *sentence*
                                   (cdr actions)
                                   *q*
                                   *stk*
                                   *token*)
                      (setq actions (cons  (cadar actions)
                                       (cdr actions))))))

             ((equal (caar actions) 'optional*)
               (if (null *sentence*)
                  (setq actions (cdr actions))
               (progn
                      (store_choicepoint
                             *sentence*
                             (cdr actions)
                             *q*
                             *stk*
                             *token*)
                  (setq actions
                                    (cons (cadar actions) actions)) )))

                   ((equal (caar actions) 'seq)
                     (setq actions (append (cdar actions) (cdr actions))))

                   ((equal (caar actions) 'either)
                       (progn
                           (dolist (acts (cddar actions) t)
                            (store_choicepoint
                                          *sentence*
                                          (cons acts (cdr actions))
                                          *q*
                                          *stk*
                                          *token*))
                           (setq actions (cons (cadar actions)
                                            (cdr actions)))))

                  ((equal (caar actions) 'parse)
                       (progn
                           (setq node (cadar actions))
                           (setq $last (newsym node))
                           (set (make$ node) $last)
                           (pushtoken $last)
                           (push *q*)
                           (setq *q* nil)
                           (setq actions (cons (list 'done
                                          $last)
                                            (cdr actions)))
                           (setq actions (cons (eval node)
                                            actions))))

                  ((equal (caar actions) 'done)
                   (progn
                     (setq temp (cons (poptoken) *q*))
                     (setq *q* (pop))
                     (setq *q* (queue temp *q*))
                     (setq actions (cdr actions))
                     (dolist (x *sem-end* t) (funcall x))
                     (setq *sem-end* nil)))

                 (t (throw 'exit t))

                );end cond
                (if (eval debug) (break "atn-main endloop"))
            );end     loop
          );end      catch
   );end           progn
 );end             defun atn-main


;;; define a semantics function --- what does a parsed word mean ?
;;; a function or action associated with the word to give it meaning.

(defmacro def-semantics (name code)
 `(setq ,name
       (list (list 'lambda nil ',code))))

;;; the intermediator between syntax and semantics functions
;;; "when" indicates when the semantic function associated with "where"
;;; should be evaluated.
;;; three possibles:
;;;   immediate, end (of the current constituent), and
;;;   wait (until a semantics immediate call on the current constituent)

(defmacro semantics (when where)
 `(cond ((equal ',when 'end) (setq *sem-end* (cons ',where *sem-end*))t)
      ((equal ',when 'immediate) (funcall ,where)
                               (dolist (x *sem-wait* t) (funcall x))
                               (setq *sem-wait* nil) t)
      ((equal ',when 'wait     ) (setq *sem-wait* (cons ',where *sem-wait*))t)
       (t nil)))

;;; add 1st argument to the sense associated with the constituent.
;;; if sense not there already create it.
;;; add-sense also adds connector to two or more

(defmacro add-sense (name code &aux temp)
 `(progn
  (setq temp (d-the sense of ,name))
  (break "name and temp")
  (cond ((null temp) (setq temp ',code))
       (t (cond ((listp (car temp)) (setq temp (cons ',code temp)))
               (t (setq temp (cons ',code (list temp)))))))
  (break "name and temp")
  (:= sense of ,name temp)
  t))

;              Dimension  default              other values

(def-featuren-number   (3s)          1s 2s 1p 2p 3p)
(def-featurev-number   (1s 2s 1p 2p 3p)  3s)
(def-featuretense      (tenseless)  present past progressive pastp)
(def-featuremood       (statement) question command)
(def-featuredative          (no)         yes)
(def-featuresense      (nil)        anything)
(def-featurereferent   (nil)        anything)


(def-net s-para                          ; A paragraph is one or
 (seq  (parse s-maj)                          ; more sentences
      (optional* (parse s-maj))))

(def-net s-maj                                ; A sentence is
 (seq        (either                                ; either
      (:= mood of $s-maj 'statement)            ; a statement

      (seq (peek verb)               ; or a command
         (== (the tense of $last)
                'tenseless)
            (insert (the word you))             ; you insertion rule
            (:= mood of $s-maj 'command))

        (seq (:= mood of $s-maj                ; or a question
                'question)
            (optional (seq (category wh)
                        (save-last)))    ; wh-movement rule

            (optional (seq (category aux)
                        (parse np)
                        (drop (the aux of $s-maj :name)) ;Aux-inversion rule
                        (drop (the np of $s-maj :name))))))
        (parse s)                       ; s-maj -> s ...
        (optional (parse fp))))



(def-net s (seq  (parse np)
            (parse vp)
            ))

(def-net fp (optional (either (category exclamation) (category question))))

(def-net np
 (either
    (seq  (optional  (category det))
          (optional* (category adj))
          (either   (category noun)
                    (category proper-noun)
                    (category pronoun)))
    (either (category proper-noun)
          (category pronoun)
          (seq  (optional (category det))
                (optional* (category adj))
                (category noun)
                (optional* (parse pp)))
          (use))))

(def-net vp
 (seq
  (optional (category aux))
  (category verb)
  (optional
    (seq  (== (the dative of $last) 'yes)
          (parse np)
          (parse np)
          (drop (the-first np of $vp :name))
          (insert (the word to))
          (drop (the np of $vp :name))
    ))
  (parse np)
  (optional* (parse pp))))
```

```
(def-net pp
  (seq (category prep) (parse np)))

(addword story nil noun)
(addword book nil noun)
(addword ball nil noun)
(addword you nil pronoun)
(addword will nil aux)
(addword give nil verb)
(:= tense of give 'tenseless)
(:= dative of give 'yes)
(addword gave nil verb)
(:= dative of gave 'yes)
(addword threw nil verb)
(:= dative of threw 'yes)
(addword told nil verb)
(:= dative of told 'yes)
(addword the  nil det)
(addword a    nil det)
(addword to   nil prep)
(addword jack nil proper-noun)
(addword Mary nil proper-noun)

(setq s1 '(jack gave mary the book))
(setq s2 '(will jack give mary the book))
(setq s3 '(give mary the book))

(atn s1 '(parse s-maj))
(print s1)
(prttree *q*)
(print (analyze 's-maj))
```

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information System Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Director, Information Systems (OP-945) Office of the Chief of Naval Operations Navy Department Washington, D.C. 20350-2000	1
4.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	2
5.	Superintendent, Naval Postgraduate School Computer Technology Programs, Code 37 Naval Postgraduate School Monterey, California 93943-5000	1
6.	Professor Robert B McGhee, Code 52Mz Computer Science Department Naval Postgraduate School Monterey, California 93943-5000	3
7.	Professor Michael J. Zyda, Code 52Zk Computer Science Department Naval Postgraduate School Monterey, California 93943-5000	2
8.	LCDR Clarence Coffman COOPMINERON22 Bldg. RTC1 Naval Base Charleston, South Carolina 29408-5900	2
9.	LT Corinne M. McConkle Fleet Numerical Oceanographic Center Monterey, California 93943-5005	3

10.	Commandant of the Marine Corps Code TE 06 Headquarters, U.S. Marine Corps Washington, D.C. 20360-0001	2
11.	Captain Andrew H. Nelson 1006 Leahy Rd. Monterey, California 93940	3
12.	Artificial Intelligence Center HQDA, OCSA ATTN: DACS-DMA Pentagon, RM 1D659 Washington, DC 20310-0200	1
13.	Deputy Commanding General Marine Corps Research Development and Acquisition Command (Code SSR) Quantico, Virginia 22134	2
14.	Director Regional Automated Services Center Camp Pendleton California, 92055 ATTN: Ray Gulath	1
15.	Professor Neil Rowe, Code 52Rp Computer Science Department Naval Postgraduate School Monterey, California 93943-5000	1
16.	US Army Combat Developments Experimentation Center (USACDEC) ATTN: W.D. West Fort Ord, California 93941	1
17.	United States Military Academy Department of Geography & Computer Science ATTN: Major R.F.Richbourg West Point, New York 10996-1695	1