NAVAL POSTGRADUATE SCHOOL
Monterey, California

AD-A218 336

THESIS

THREE-DIMENSIONAL ROUTE PLANNING
FOR A CRUISE MISSILE
FOR MINIMAL DETECTION BY OBSERVER

by

Lawrence R. Wrenn III

June 1989

Thesis Advisor: Neil C. Rowe

90 02 26 196

REPORT DOCUMENTATION PAGE

1a Report Security Classification **UNCLASSIFIED**	1b Restrictive Markings
2a Security Classification Authority	3 Distribution Availability of Report
2b Declassification/Downgrading Schedule	Approved for public release; distribution is unlimited.

4 Performing Organization Report Number(s)	5 Monitoring Organization Report Number(s)

6a Name of Performing Organization Naval Postgraduate School	6b Office Symbol *(If Applicable)* 52	7a Name of Monitoring Organization Naval Postgraduate School
6c Address *(city, state, and ZIP code)* Monterey, CA 93943-5000		7b Address *(city, state, and ZIP code)* Monterey, CA 93943-5000
8a Name of Funding/Sponsoring Organization	8b Office Symbol *(If Applicable)*	9 Procurement Instrument Identification Number

8c Address *(city, state, and ZIP code)*	10 Source of Funding Numbers

Program Element Number	Project No	Task No	Work Unit Accession No

11 Title *(Include Security Classification)*
THREE-DIMENSIONAL ROUTE PLANNING FOR A CRUISE MISSILE FOR MINIMAL DETECTION BY OBSERVERS

12 Personal Author(s)
Wrenn, Lawrence R. III

13a Type of Report Master's Thesis	13b Time Covered From To	14 Date of Report *(year, month, day)* June 1989	15 Page Count 221

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

17 Cosati Codes			18 Subject Terms *(continue on reverse if necessary and identify by block number)*
Field	Group	Subgroup	Artificial Intelligence; LISP; Three-dimensional Search, Path Planning, Route Planning, Search.

19 Abstract *(continue on reverse if necessary and identify by block number*

We present an algorithm for finding optimal three-dimensional paths above polyhedral models of terrain using a technique we refer to as "random-ray". Contiguous sequences of homogeneous airspace volumes are generated using constraints of probability-of-detection and aerodynamic-flight models. The flight costs are calculated as in actual mission planning using time, distance, airspeed, and fuel flow. We then try semi-random directions (rays) from the starting point, turning in accordance with Snell's Law at maneuver points (points between volumes). If we ever do not enter the previously specified next volume, we make random adjustments to the ray (in, out, up, or down) with respect to the center of the facet between the two volumes, until either the path will enter the correct next volume or we determine it is impossible. The performance of our random-ray technique is an improvement over an earlier approach using local optimization. We have also implemented a movable display on a graphics workstation, to allow the user the ability to view the terrain and paths from any angle.

20 Distribution/Availability of Abstract [X] unclassified/unlimited ☐ same as report ☐ DTIC users	21 Abstract Security Classification UNCLASSIFIED
22a Name of Responsible Individual Prof. Neil C. Rowe	22b Telephone *(Include Area code)* (408) 646-2462	22c Office Symbol Code 52Rp

Three-Dimensional Route Planning for A Cruise Missile
For Minimal Detection By Observers

by

Lawrence R. Wrenn III
Major, United States Marine Corps
B.S., Virginia Military Institute, 1976

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1989

Author: _____
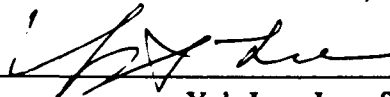
Lawrence R. Wrenn III

Approved by: _____

Neil C. Rowe, Thesis Advisor

_____

Yuh-Jeng Lee, Second Reader

_____

Robert B. McGhee, Chairman
Department of Computer Science

_____

Kneale T. Marshall
Dean of Information and Policy Sciences

ii

ABSTRACT

We present an algorithm for finding optimal three-dimensional paths above polyhedral models of terrain using a technique we refer to as "random-ray". Contiguous sequences of homogeneous airspace volumes are generated using constraints of probability-of-detection and aerodynamic-flight models. The flight costs are calculated as in actual mission planning using time, distance, airspeed, and fuel flow. We then try semi-random directions (rays) from the starting point, turning in accordance with Snell's Law at maneuver points (points between volumes). If we ever do not enter the previously specified next volume, we make random adjustments to the ray (in, out, up, or down) with respect to the center of the facet between the two volumes, until either the path will enter the correct next volume or we determine it is impossible. The performance of our random-ray technique is an improvement over an earlier approach using local optimization. We have also implemented a movable display on a graphics workstation, to allow the user the ability to view the terrain and paths from any angle.

Accession For

NTIS CRA&I	X
DTIC TAB	☐
Unannounced	☐
Justification	

By
Distribution/
Availability Codes

Dist	Avail and/or Special
A-1	

DTIC COPY INSPECTED 5

TABLE OF CONTENTS

I. INTRODUCTION

Prior planning has always been the key to success of any military mission. With advance notice, all possibilities can be thoroughly examined and the best possible choices made. Unfortunately, time is often at such a premium in last-minute strike planning that all avenues of attack are not fully exploited. Furthermore, in today's changing world we are continually confronted by changes in policy, strategy and assets. A domino effect from upper echelons causes constant turmoil in preplanned strike packages. A faster, more efficient way of route planning, and systems that can detect the enemy's weakest avenues of approach are needed. Such systems could also be used to identify our own deficiencies and allow for reinforcements.

Our work at the Naval Postgraduate School (NPS) has led to new methods of path planning using artificial intelligence that are more suited for near-real-time path planning. This new approach, combined with proven algorithms from artificial intelligence about searching through sets of possible solutions to find particular "best solutions", has allowed faster results to certain path-planning problems than was previously possible. A program of this type could greatly aid aircrews in their mission planning, or add flexibility to path-planning for cruise missiles. Once in general use, where basic data is already loaded, paths can be planned by inputting the start and goal points and how many paths we wish to consider and the computer program, suitably optimized, should be able to give us answers within minutes for

1

many interesting problems. Considering that it takes an average of two days to fully plan manually a single route into hostile territory, this is a great savings. The computer does not forget special considerations under pressure and it can give us all possible solutions, not just the obvious ones.

Several programs have come close to what we are looking for but have fallen short for various reasons. Flight planning programs while giving accurate flight information have no graphics display of actual terrain and no path planning. Theoretical development of three-dimensional path planning has occurred recently but nothing is available for true aircraft models when the searching to reduce the probability of detection. Much work has been done with terrain displays at NPS but none include path planning or aircraft realism.

Our approach is to implement all the ideas of the last paragraph into one system. We have extended David Lewis's thesis [Ref. 1] to include aircraft realism in the cost analysis for the search algorithm, and used a new technique for optimization we refer to as "random- ray optimization". Additional work was done on visualization of the search algorithm and graphic display of the final results.

The remaining sections of this thesis describe the work mentioned above. Chapter II is a brief overview of previous work, including a description of other programs available. For a more through understanding it is suggested that David Lewis's thesis [Ref. 1] be reviewed. Chapter III discusses assumptions made about our aircraft (a pseudo-cruise-missile), while Chapter IV is the detailed account of algorithms implemented. The results of the implementation are given in Chapter V,

and finally in Chapter VI we discuss conclusions, known problems, and recommendations for further study.

II. BACKGROUND

A. FLIGHT PLANNING PROGRAMS

Flight planning is just one small part of an overall mission plan. The full mission plan may contain other items such as target coordination, secondary targets, and route planning. Route planning is related to flight planning in that the route must be known in order to compute fuels and attack times to complete flight planning.

1. OPARS

Optimal Path Aircraft Routing System (OPARS) and similar flight planning programs are used extensively by the military, particularly for long haul type aircraft such as C-5's, C-141's and C-130's.[Ref. 2] OPARS has flight characteristics for the appropriate aircraft available and receives input from Fleet Numerical Oceanography Center in Monterey, California on items such as forecast winds and temperatures. The program can plan flights from point to point where the points are known points such as TACAN stations or fixes with known latitude and longitude, giving priority to fuel consumption and avoidance of high winds. The algorithm used for OPARS is a depth-first search which generates a series of possible paths, and from this series, the best path is chosen.

OPARS is a prior-planning tool that gives information from the latest available data with the output accurate enough to be used in-flight as a check

against how far ahead or behind schedule the aircraft is. It is predominately a high-altitude tool, and uses known points as discussed in the previous paragraph to plan to. Our goal is to evolve something that uses points that it defines and can be used at low altitude for detection avoidance.

2. Calculator Aided Performance Planning System

Special-purpose hand-held calculators have been used for flight planning for tactical aircraft such as E-2C's, E-2B's and A-6's [Ref. 3] and [Ref. 4]. Prior planning done on these small devices is much simpler and less error-prone then using graphical charts [Ref. 5]. They can be used in the actual aircraft when a change in plan is desired or required because of some emergency. The data contained in these machines is extremely accurate and is a compilation of the data contained in the NATOPS Flight Manual[1]. While some of this information is contained in the pocket version of the NATOPS carried in the aircraft, it is not complete.

The main problem with this device is that data entry is no trivial matter (twenty or thirty steps on a number key pad). There is no way to store the input in flight planning, and if there is a change, go back in and change a few items. You must also know where you are going and how you are going to get there. But part of these programs are usable in our program, the flight data formulas. These

---

[1]NATOPS is the Naval Air Training and Operating Procedures Standardization Program which contains vital information for the safe and emergency operation of military aircraft.

formulas were coded in the flight characteristics section of our program and used by the cost and evaluation section of the search algorithms.

3. Others

Since 1984, low cost and improved performance of microcomputers have made it practical to develop computer-aided mission-planning-tools for use at the squadron level [Ref. 6]. Some have used the flight data from NATOPS, [Ref. 6], and others have required input of fuel consumptions at every stage of the flight. Most have the ability to store and change the mission plan, but all require that the route be known beforehand.

B. THREE-DIMENSIONAL PATH PLANNING

1. Division Of Search Space

The speed at which any computer can solve a search problem is dependent on the search algorithm used and the size of the search space. If we were to divide a cube of side one thousand into cubes of side ten, we would have one million cubes to search through to find a connected path from some start cube to a goal cube. If we were to make our cubes larger to side twenty five, the search space is reduced to sixty four thousand blocks. This is a large reduction in size of the search space but it is still large. The complexity of a search problem is directly proportional to the search space when it comes to allocating time and space resources for a simple non-heuristic search [Ref. 7].

But the search space need not be subdivided uniformly. Earlier work [Ref. 1] used the physical features of polyhedrally-modeled terrain for the first division of the airspace. We will refer to these divisions of airspace as volumes as they are bounded on all sides and each will have a homogeneous property of some visibility constant. Vertical planes were constructed above all ridges forming convex volumes so that from any point in a volume every other point is visible. Once observer data is added to the problem, these convex volumes are further divided into visibility volumes by passing planes from the observer through the peaks of all ridges. Each resulting volume has an associated probability of detection from each observer that it is visible to. If one volume is visible to several observers, its probability of detection is calculated assuming probabilistic independence.

2. Path Planning Algorithm

An A* search is used to produce a connected path from center-of-volume to center-of-volume in [Ref 1]. The A* search was chosen to find good sequences of volumes likely to enclose the optimal path because A* allows the use of an agenda, an evaluation function, and a cost function. The [Ref. 1] program used a cost function that took into account climb, dive, and amount of turn, all multiplied by some function of the probability of detection. The evaluation function was calculated in a similar manner.

We still considered this method of search the best, but we made modifications to the cost and evaluation functions. These functions have been

altered to reflect True Air Speed (TAS) of the missile, weight, Fuel Flow (FF) and time spent in a particular region. This was done to ensure a more realistic aerodynamic model rather then the simple percentages used in [Ref. 1] and to allow for specific aircraft data to be encoded at a later date.

3. Optimization of Paths

Once the volume sequences are found in the program of [Ref. 1], initial paths are generated from center-of-facet to center-of-facet of the polyhedron through the search space. This means that the paths may go a considerable distance out of their way if only a corner of the volume need be passed through. [Ref. 1] used a modification to Snell's Law to move the facet intersection points to try to minimize the error in the Snell's Law equation. This is repeatedly applied to a path until the desired tolerance is obtained. The problem encountered was that the process would get stuck on local optimization. This happens at irregular intervals and can therefore not be anticipated and corrected.

C. WORK IN COMPUTER GRAPHICS DISPLAYS

Recent work at NPS has explored the use of graphic displays to present real terrain from elevation data. One of the most recent of these reads in the terrain data base and allows the user to select a segment of this for a three-dimensional view of the terrain from various platforms such as jeeps, trucks, tanks and even a missile [Ref. 8]. Control inputs for the missile are via dials for altitude, speed and direction. This would be good for output from our program, but this software at present does not display the missile path nor is there any intelligent path planning.

8

III. APPLICATION AND ASSUMPTIONS

A. REAL WORLD PROBLEMS

Flight planning is a tedious, calculation-intensive and error-prone process. Many hours of planning can be wiped away by a simple change in commands from higher authority or new intelligence data on the location of a missile or radar site. Some target areas are so saturated with defenses that there exists no good way to attack, only the least hazardous. In these situations it is difficult for any human to rationally plan a route into a target he knows he may never come out of. Likewise, when planning for the cruise missile to destroy a site that will open a path that is critical for other aircraft to take, it is essential that the path chosen for this missile is survivable.

This type of planning can become an overpowering task. For this and many other reasons, U.S. Naval Instructions require that aircrews be given the opportunity for eight hours of rest prior to flying. In some cases this is not possible, so anything that will help lighten the workload is a big plus. Powerful computer programs can help with the mass of calculations required for the single flight of an aircraft or cruise missile. A program of this type can be used in the strategy planning room at the Wing or Battle Group level, or by the individual pilot at the squadron level.

Higher headquarters are constantly playing "what if" games in contingency planning. Furthermore, every time there is a change in situation, planners must review all the preplanned strikes to ensure that they have covered all the changes in targets, defenses and missions that need to be addressed. Similarly, commanders must review our own defensive posture to ensure we have not left any open passages.

When it comes time for an actual conflict similar problems will be encountered. The need for computer simplicity and accuracy is essential. The computer can cut calculations to a fraction of the time and present many more path-planning options than could be produced by several human planners.

B.   AIRCRAFT REALISM

In order to keep this thesis unclassified, no attempt was made to obtain any classified documentation on the cruise missile. It is important however to understand what information is needed so that appropriate substitutions could be made for actual flight data at a later date.

1.   Physical Characteristics

The model of the cruise missile we used was a variant of the Tomahawk. It measures approximately 20 feet with a wing span of 8 feet 7 inches and has a diameter of 21 inches.[Ref. 9] The missile, with a full fuel load of 900 lbs (approximately 120 gallons), weights 2525 lbs.[Ref. 10] The engine used is a turbofan developed by Williams Research Corporation and has a designation of

F107-WR-100. This engine can produce a static thrust of 430 lbs at sea level and has a specific fuel consumption of 0.7 lb/lb-hr.[Ref. 11]

All the articles read on the cruise missile indicate that the planned cruise speed is around 450 kts. This can be increased or decreased depending on the importance of achieving minimum detection or increasing range. The speed we will assume in this program is 450 kts. As shown in Figure 3-1, the turn characteristics are such that the missile will lead a turn by an amount sufficient to arrive wings level on an outbound course directly between the turn point and the next point.[Ref. 12] This turn has a radius of 5 nm and is accomplished in 1G flight so as to not bleed any excess energy or require any radical power changes.

Fuel consumption data versus vehicle weight is listed in column form in Table 3-1 and shown in graphic form in Figure 3-2. The data was derived from graphs modeled

TABLE 3-1. Cruise Missile Weight vs Fuel Flow

Missile Weight	Fuel Remaining	Fuel Flow
2525	1225	350
2275	975	325
2025	725	300
1775	475	285
1525	225	275

after the cruise performance charts for the Grumman A-6 aircraft, [Ref. 13]. This data is for straight and level flight assuming an average fuel flow of 270 lb/hr which will maintain the required 450 kts. Equation 3-1 gives the computation for fuel remaining (X) against fuel flow.

$$\text{Fuel Flow} = -1.6e\text{-}10 * X^4 + 4.3733e\text{-}7 * X^3 - 3.566e\text{-}4 * X^2 + 0.1530066 * X + 254.05494 \qquad (3\text{-}1)$$

11
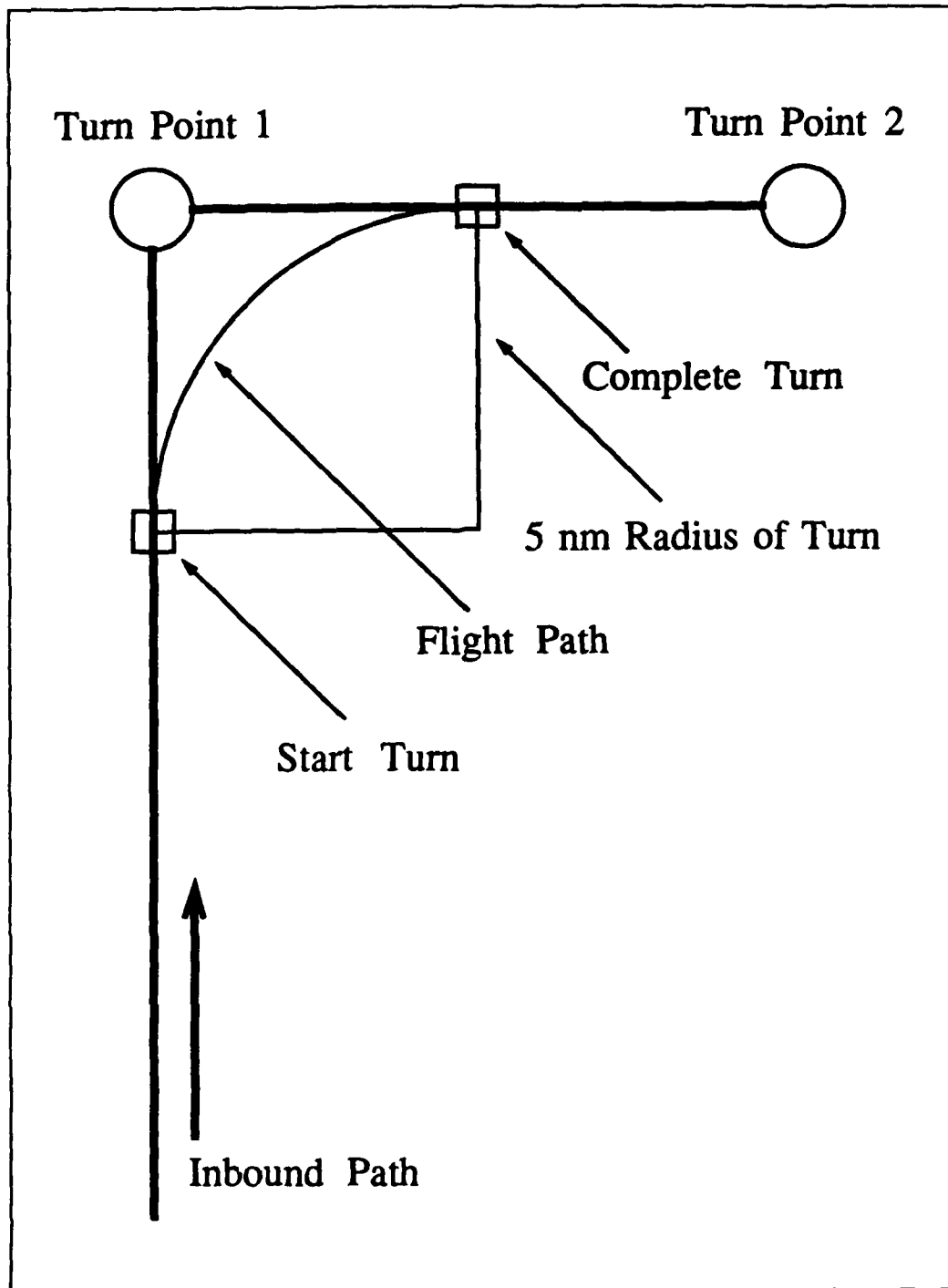
Figure 3-1. Cruise Missile Turn Characteristics

FUEL WEIGHT vs FUEL FLOW

To Maintain 450 Kts
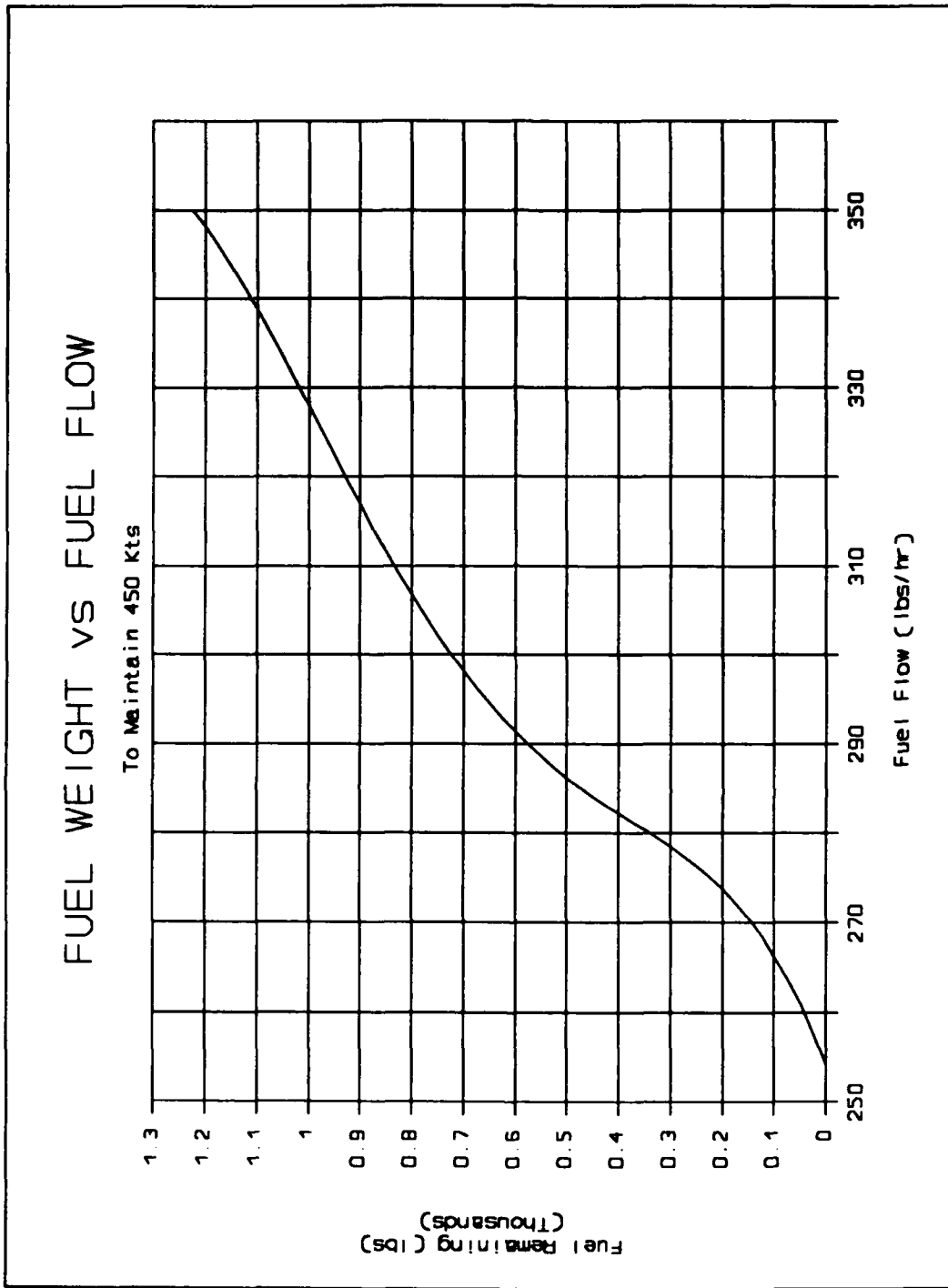
Figure 3-2. Cruise Missile Weight vs Fuel Flow

13

At this rate the 900 lbs of useable fuel will allow a maximum distance of 1500 nm. The formula for equation (3-1) and (3-2) were obtained by using selected points of graphs and using a polynomial curve approximation of degree four and three [Ref. 14].

For climbs and dives we use a different set of formulas modeled after aerodynamic theory from [Ref. 15] and personal experience. For a climb up to 20 degrees and a dive of less then 10 degrees we assume the missile adjusts power to maintain 450 kts. The fuel flow for this power adjustment is given by the equation

$$\text{Fuel Flow} = 0.01628787 * X^3 + 0.1037878 * X^2 + 21.40909 * X + 300 \tag{3-2}$$

where X is the angle of climb and is depicted graphically in Figure 3-3. For dives steeper then -10 degrees the missile will increase speed and when it returns to level flight the engine will remain at idle until such time that the aircraft decelerates to 450 kts. For a climb greater then 20 degrees the rate at which airspeed will be lost is

$$\text{Loss Rate} = 3\text{kts}/(\text{climb degrees} - 20)/\text{min} \tag{3-3}$$

and the rate at which this airspeed can be recovered is

$$\text{Recovery Rate} = 50\text{kts/min} \tag{3-4}$$

which if the speed is decreased to 200 kts it will require 5 minutes to accelerate back to 450 kts.

14
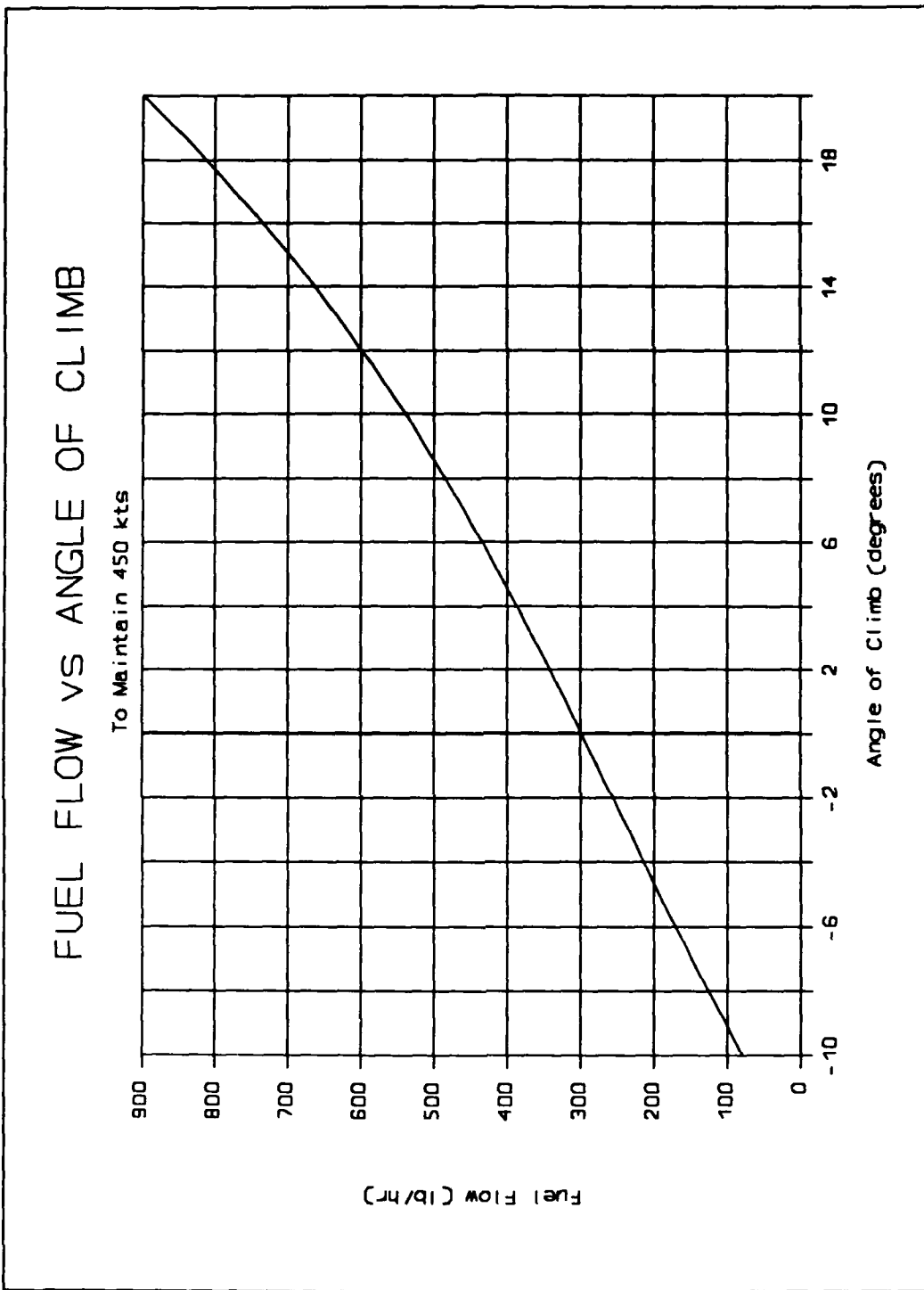
Figure 3-3. Fuel Flow vs Angle of Climb

15

IV. IMPLEMENTATION

A. OVERVIEW

1. System Requirements

The path-planning part of the thesis was implemented on a Texas Instrument (TI) Explorer II LISP machine with 16 megabytes of memory and 60 megabytes of virtual memory. The code is written in Common LISP and makes extensive use of the LISP Flavor System. The program forerunner of D. Lewis, [Ref. 1], was written in LISP because of the advantages in speed, numerical accuracy and sophisticated data-structure management, and we have continued with LISP for these same reasons [Ref. 1:64]. Since the project was started there have been two upgrades in the operating system with no problems or re-coding required. The program will also run on a TI Explorer I LISP machine if sufficient memory is available but at a large increase in execution time.

2. Main Program Parts

The program can be broken into three main sections (Figure 4-1): terrain input and processing, observer input and processing, and path planning and optimization. The first two sections have not changed from [Ref. 1:77] and will not be discussed here. The final section has significant differences, due particularly to the use of a quite different technique, "random-ray optimization". It also has one

16
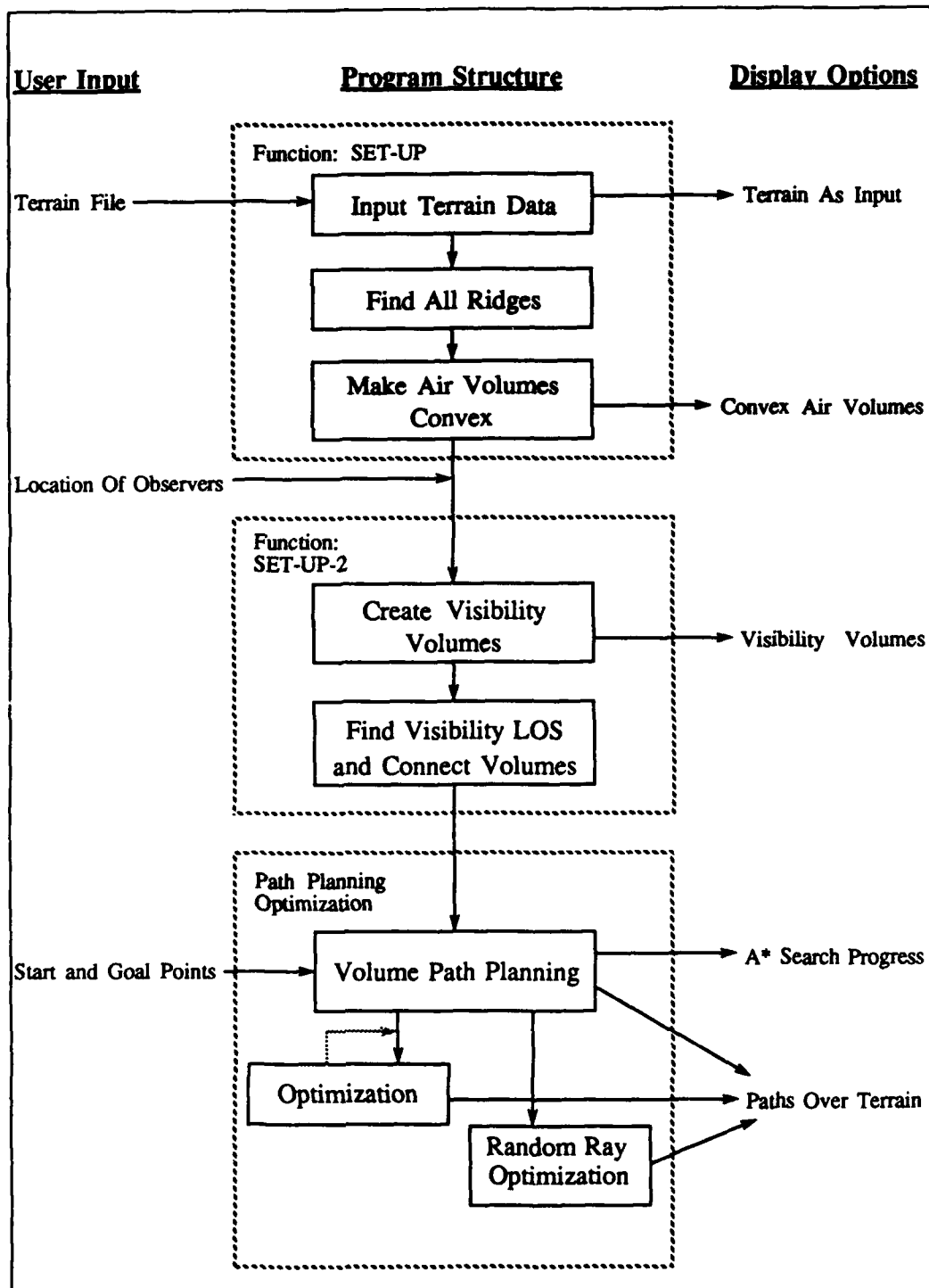
Figure 4-1. Block Diagram of Program Structure

new display option which allows the user to observe the best agenda item as it is being changed by the A* search.

3.   Output Data

As in [Ref. 1], the paths our program finds are made up of linear pieces beginning with the volume that contains the start point and ending with the volume that contains the goal point. This path is further defined by the individual facets connecting the volumes and the turn points on the facets that when connected will form a path from the start point to the goal point.

The path can be the input of functions that will give specific data about it such as length, travel time, visibility along each line segment, amount of fuel used for each segment, and total fuel used. From this data paths can be compared, and determinations made as to which path is best suited for the particular mission. A sample of this output is given in Table 4-1.

4.   Contributions of Others

The code written by Lewis has been used extensively.[Ref. 1] Little or no changes have been made to [Ref. 1] code up through the path planning section. The section of code for the A* search is still used intact but the cost and evaluation functions have been completely rewritten. For some paths there are no random-ray solutions so [Ref. 1] code for optimization of a path was left intact. This gives us a way to handle all situations.

A set of moving picture display functions developed by Dr. Sehung Kwak were added to give the ability to visualize the A* search as it

TABLE 4-1. Sample "Jet Log" Type Output

```
> (path-data 'Ipath00321)
            Leg    Total  Leg    Total  Leg    Fuel    Vol    PD     Leg
Point       Time   Time   Dist   Dist   Fuel   Remain  PD     Cost   Cost
(10.0 0.0 300.0)
            0.0    0.0    0.0    0.0    0.0    1500    -      -      -
(469.23077 300.0 601.53845)
            73.1   73.1   548.5  548.5  367.0  1133.0  0.070  512.0  879.0
(111.53846 500.0 346.92307)
            54.6   127.8  409.8  958.3  272.1  860.9   0.070  382.5  654.6
(111.53846 462.5 287.30768)
            5.0    132.8  37.5   995.8  24.7   836.2   0.070  35.0   59.7
(586.53845 567.5 262.3077)
            64.9   197.6  486.5  1482.3 324.2  512.0   0.000  0.0    324.2
(700.0 700.0 380.0)
            23.3   220.9  174.4  1656.8 116.8  395.1   0.000  0.0    116.8
(420.0 852.5 505.0)
            42.5   263.4  318.8  1975.6 213.1  182.0   0.000  0.0    213.1
(990.0 990.0 990.0)
            78.2   341.6  586.3  2561.9 393.1  -211.1  0.070  547.3  940.3
Total cost of this path -   3187.8
minimum PD cost -      0.0
maximum PD cost -    547.3
average PD cost -      9.3
3187.77618788756l3d0
> (path-data 'Ipath00341)
            Leg    Total  Leg    Total  Leg    Fuel    Vol    PD     Leg
Point       Time   Time   Dist   Dist   Fuel   Remain  PD     Cost   Cost
(10.0 0.0 300.0)
            0.0    0.0    0.0    0.0    0.0    1500    -      -      -
(70.92308 300.0 328.15384)
            40.8   40.8   306.1  306.1  204.2  1295.8  0.070  285.7  489.9
(73.65098 310.91418 325.41046)
            1.5    42.3   11.2   317.4  7.5    1288.3  0.070  10.5   18.0
(276.87296 412.58145 300.40717)
            30.3   72.6   227.2  544.6  151.4  1136.9  0.070  212.1  363.5
(586.53845 567.5 262.3077)
            46.2   118.8  346.3  890.9  230.7  906.3   0.000  0.0    230.7
(509.11267 700.0 375.13834)
            20.5   139.2  153.5  1044.3 102.8  803.4   0.000  0.0    102.8
(104.55399 1000.0 500.9108)
            67.2   206.4  503.7  1548.0 336.3  467.1   0.000  0.0    336.3
(990.0 990.0 990.0)
            118.1  324.5  885.5  2433.5 592.5  -125.4  0.070  826.5  1419.0
Total cost of this path -   2960.2
minimum PD cost -      0.0
maximum PD cost -    826.5
average PD cost -      9.1
2960.188312228768d0
```

progressed.[Ref. 16] This code originally written to display one graphics window and one moving object was altered to display several windows and multiple objects.

5. Data Structures

The data structures have not changed in any area except for the agenda in path planning. The agenda now contains the fuel remaining at each turn point so the next leg's fuel flow can be calculated, and the last airspeed must be retained so we know where to start our calculations for time and distance.

6. Deviations From Total Path-Planning

It was initially intended to include all aspects of path-planning in our new path-planning method but due to time-constraints and the complexity of the problem planning around obstacles, except for the minor cases, was left out. Obstacle-traversal by paths would have required additional algorithms in the A* search and the optimization phase that would allow the paths to be sectioned, thus complicating matters.

B. NEW PATH-PLANNING METHODS

1. Initial Path-Planning

Path planning begins by initializing the start and goal point with INIT-POINT. These points are passed to the search function A-STAR-SEARCH or A-STAR-SEARCH-M. An additional switch has been added to these two functions and if set to true, the best path on the agenda will be displayed as the search progresses, Figure 4-2. The two upper and lower left displays clear each time a new path is made and the fourth displays all lines as they are generated. The final display will show all the final paths from start to goal.

20
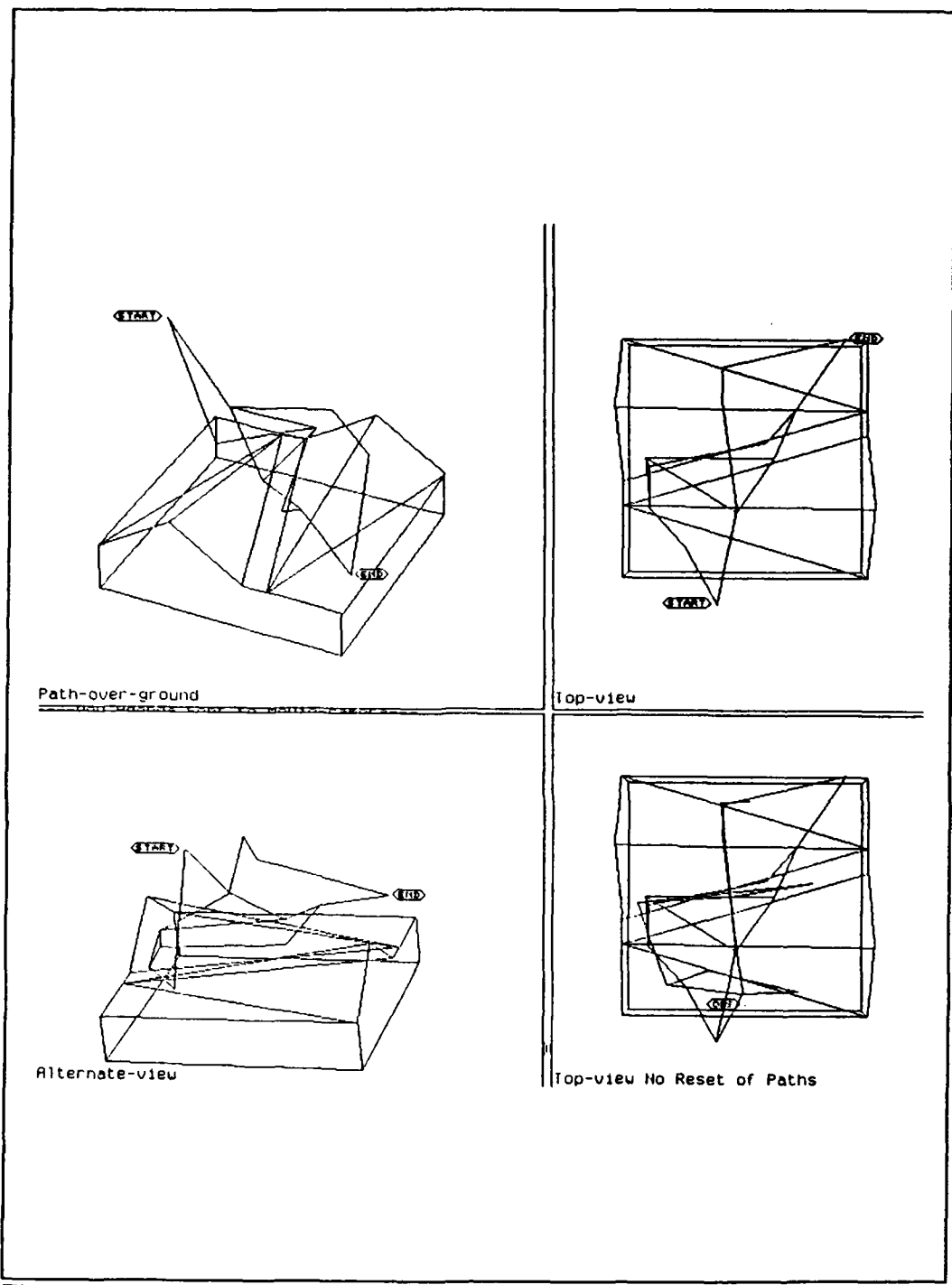
Figure 4-2. Search Display

The cost function consists of two parts, a probability of detection (PD) cost and the cost of fuel used to get from one point to another. The PD cost is calculated using

$$PDcost = probability\text{-}of\text{-}detection * time\text{-}in\text{-}volumes * 100 \qquad (4\text{-}1)$$

where the extra 100 is multiplied in to weight the PDcost to an amount comparable to the basic cost. This causes a short stay in a volume of high probability of detection to be preferred over a long stay in a volume of medium probability of detection. It also forces the searcher to look for volumes with zero probability of detection. The fuel-burned cost is related to the distance flown and how much climbing and diving is done, as discussed in the previous chapter. Because of nonlinear aerodynamics, a missile or aircraft does not gain back the fuel it lost in a climb if it descends back to the same level [Ref. 17]. For this reason, paths that remain at a constant altitude are preferred.

The evaluation function is a calculation of the fuel cost from our current location directly to our goal point ignoring obstacles. No attempt was made to add in a PD cost as we do not know what volumes we will be going through or the time we will spend in them.

2. Aircraft Data

The cost and evaluation functions receive all of their input on aircraft data from the aircraft control module. Inputs to this module include the distance traveled (not just ground distance), the climb angle, the fuel remaining, and current airspeed. The program limits the fuel flow to an idle setting of 80 lb/hr which is

22

the setting in a ten degree dive and a maximum of 900 lb/hr at maximum power. The module returns the fuel burned on that leg, fuel remaining and new airspeed.

3. Random Ray Optimization

The random-ray technique is applied to the best connected sequences of volumes from start to goal that were found by the A* search. This technique for a particular volume sequence can be broken into three parts: finding a path into the final volume, adjusting the path as close to the goal point as possible, and calculating additional path details. The last part is required because due to the number of lines and points generated during the adjustment phase, minimal data is kept for each.

To start, a line, a "random-ray", is passed from the start point to the goal point. This ray is examined to determine if it passes into the specified second volume via the connecting facet (the plane connecting the two volumes). If it does not, a guess adjustment is made to the ray using an adjustment vector calculated from the actual intersect point on the extended facet to the center of the facet (Figure 4-3). The adjustment vector is multiplied by an adjustment factor (initially 125) and then added to the end point of the ray to obtain a new random-ray. These adjustments are continued, each time dividing the adjustment factor by five if the distance we are missing the facet by is increasing, until the path intersects the facet. We then calculate the outbound ray using Snell's Law (described below) and find the next line segment in the path.

23
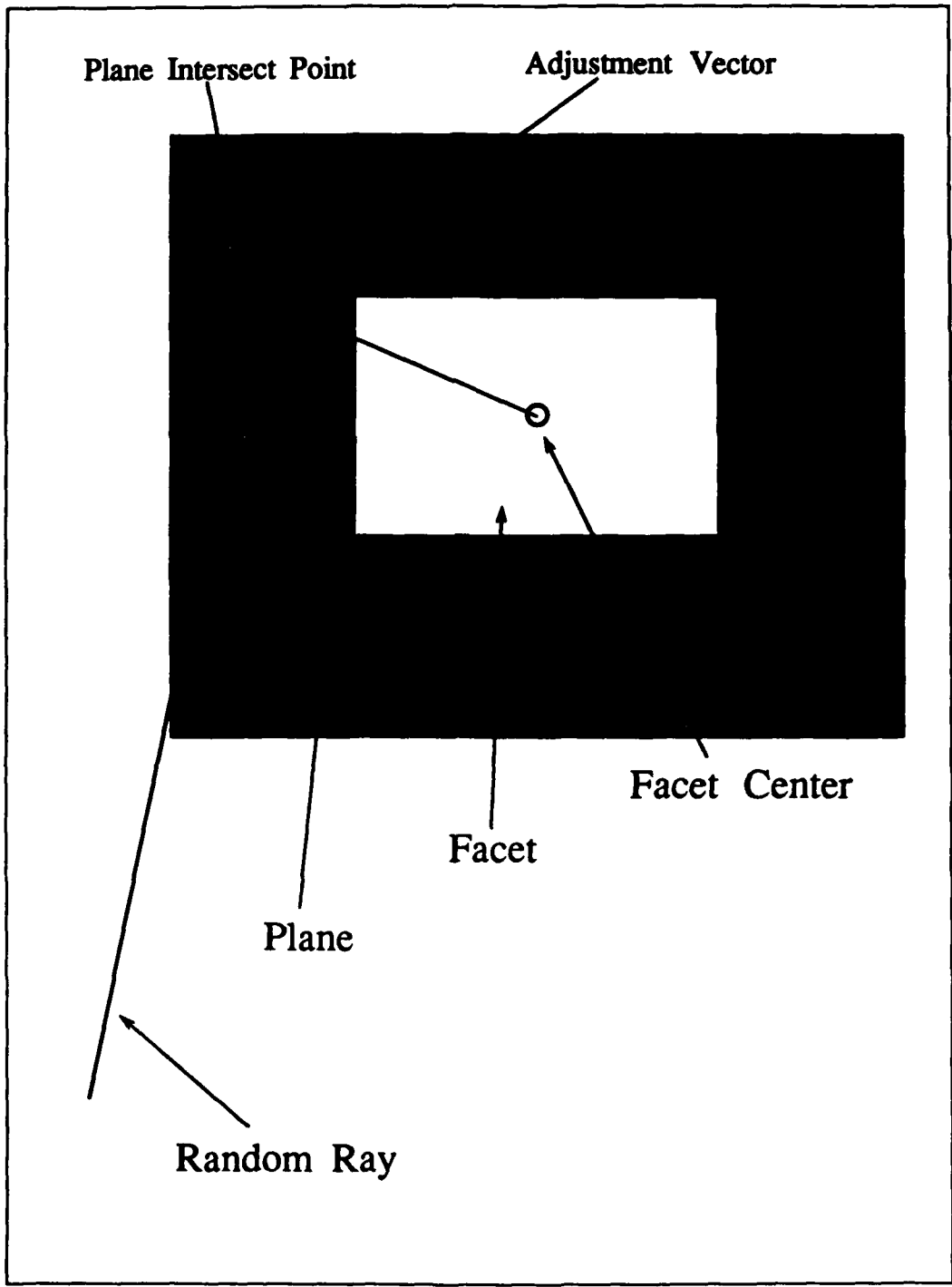
Figure 4-3. Adjustment Vector Determination

24

Calculations are made at each facet to find the outbound line that meets the criterion of Snell's Law. Formulas were developed for a known inbound ray to the facet, the plane that the facet lies in, and the amount of bending the ray must do according to Snell's Law (Figure 4-4). Assume the equation of the plane containing the facet is

$$Ax + By + Cz + 1 = 0 \qquad (4\text{-}1)$$

where (A B C) is the vector normal to the plane. Using the point of intersection of our inbound ray and (A B C) (the vector normal), we generate a line perpendicular to the facet. Once two lines are obtaied we can generate a plane, equation (4-2), contaning both.

$$A_2x + B_2y + C_2z + 1 = 0 \qquad (4\text{-}2)$$

We know the unit direction vector $(i_1\ j_1\ k_1)$ of the inbound line and are trying to find the unit direction vector $(i_2\ j_2\ k_2)$ of the outbound line. We have then three equations in these three unknowns:

$$A_2(i_2) + B_2(j_2) + C_2(k_2) = 0 \qquad (4\text{-}3)$$

$$(i_1)(i_2) + (j_1)(j_2) + (k_1)(k_2) = \sin(theta_2 - theta_1) \qquad (4\text{-}4)$$

$$i_2^2 + j_2^2 + k_2^2 = 1 \qquad (4\text{-}5)$$

$Theta_1$ is given and is the angle between the inbound ray and the facet normal (Figure 4-4). $Theta_2$ can be calculated using equation (4-6) where PD1 and PD2 are the respective volume's probability of detection.

$$Theta_2 = \arcsin(PD1 * (\sin Theta_1) / PD2) \qquad (4\text{-}6)$$

We can solve in terms of any one of i_2, j_2, or k_2 and substitute this into equation (4-5), which is easily solved using the quadratic formula. As it turns out, we need

25
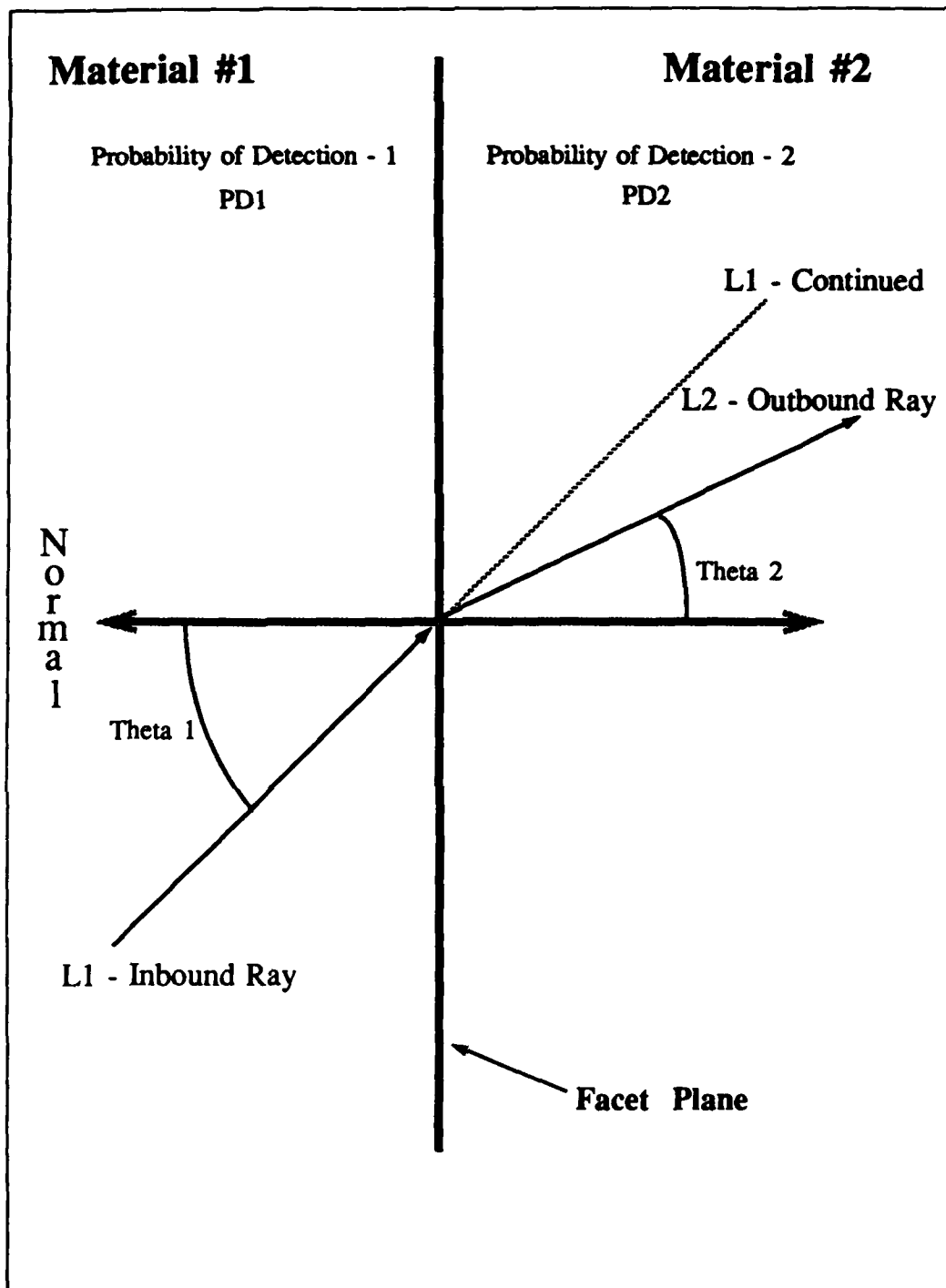
Figure 4-4. Snell's law

26

all three of $(i_2 \; j_2 \; k_2)$ to avoid a possible divide-by-zero error. We also want to avoid complex roots; if the square root of a negative number is about to be taken, we check to see if that number is approximately zero, and make it zero. If however these two errors can not be avoided, the program is terminated indicating that no random-ray solution is possible.

Once we have found the outbound ray and know the point of intersection with the next facet plane, we can construct the outbound line segment. This ray can be adjusted to hit within the facet as with the first ray. If this adjustment causes the ray to miss any of the previous facets, the adjustment is thrown out and a new guess is made. This is done for every successive facet of the volume sequence, until we intersect the facet connected to the final volume. Now our target has changed; we are now shooting for a point in space rather then a window. The adjustment technique remains the same except we make adjustments in smaller increments. Figure 4-5 shows the path generated by connecting the centers of the facets of the volumes found by the A* search, and a straight-line path from start to goal. Figure 4-6 and Figure 4-7 show adjustments needed to enter the goal volume and Figure 4-8 shows adjustment onto the goal point. An analogy of all of this is adjusting artillery fire onto a target, the only difference being that we do not know adjustment sensitivity, which varies dramatically.

Goal Point

Straight Line Path ⟶
(Does Not Abey Snell's Law
Nor Intersect Facet #5)

Facet #5

Facet #4

Facet #3

Facet #2

Facet #1

⟵ Original Facet to Facet Path
(Does Not Obey Snell's Law)

Start Point
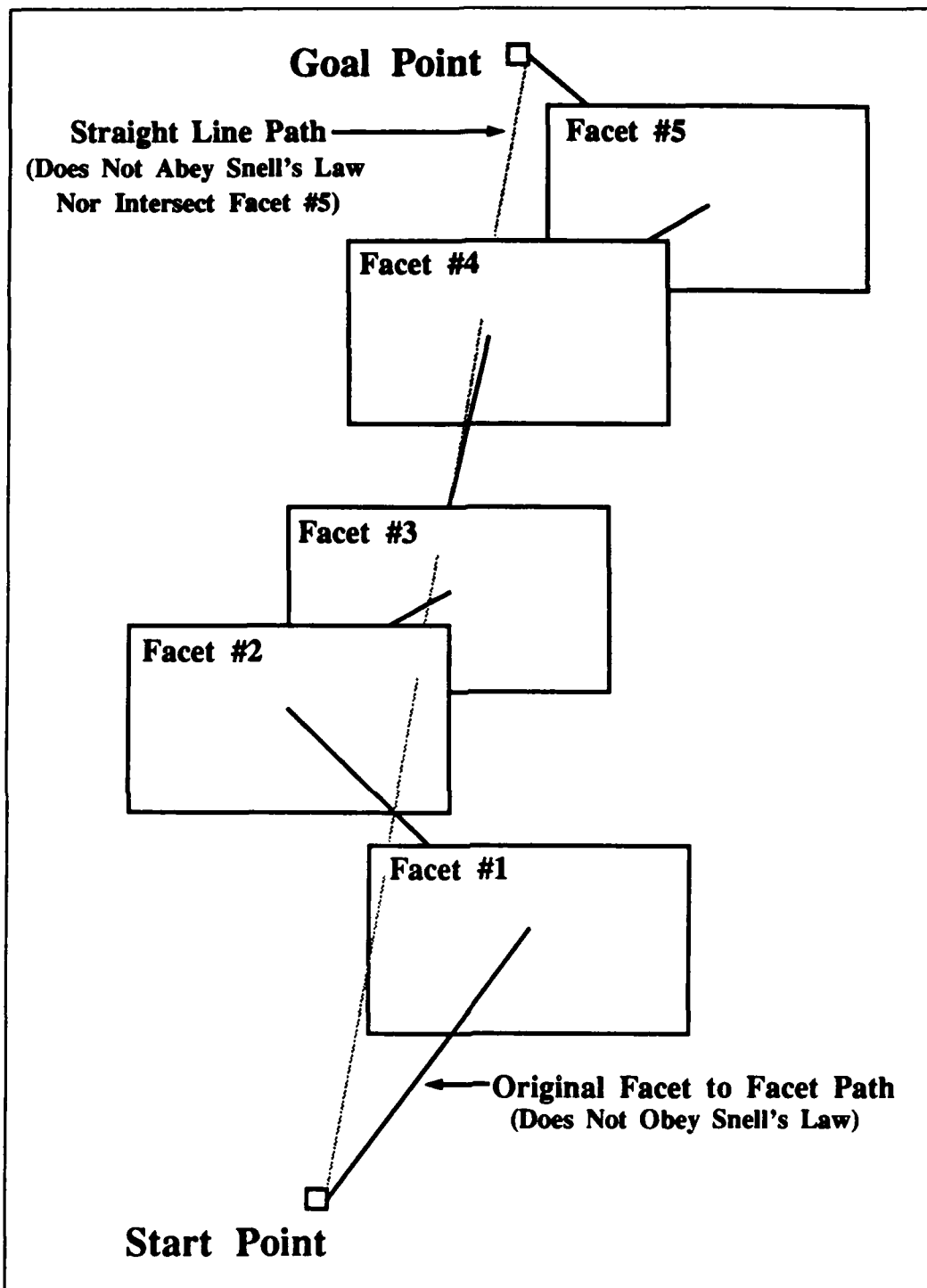
Figure 4-5. Original Path to Goal and Line-Of-Sight Path
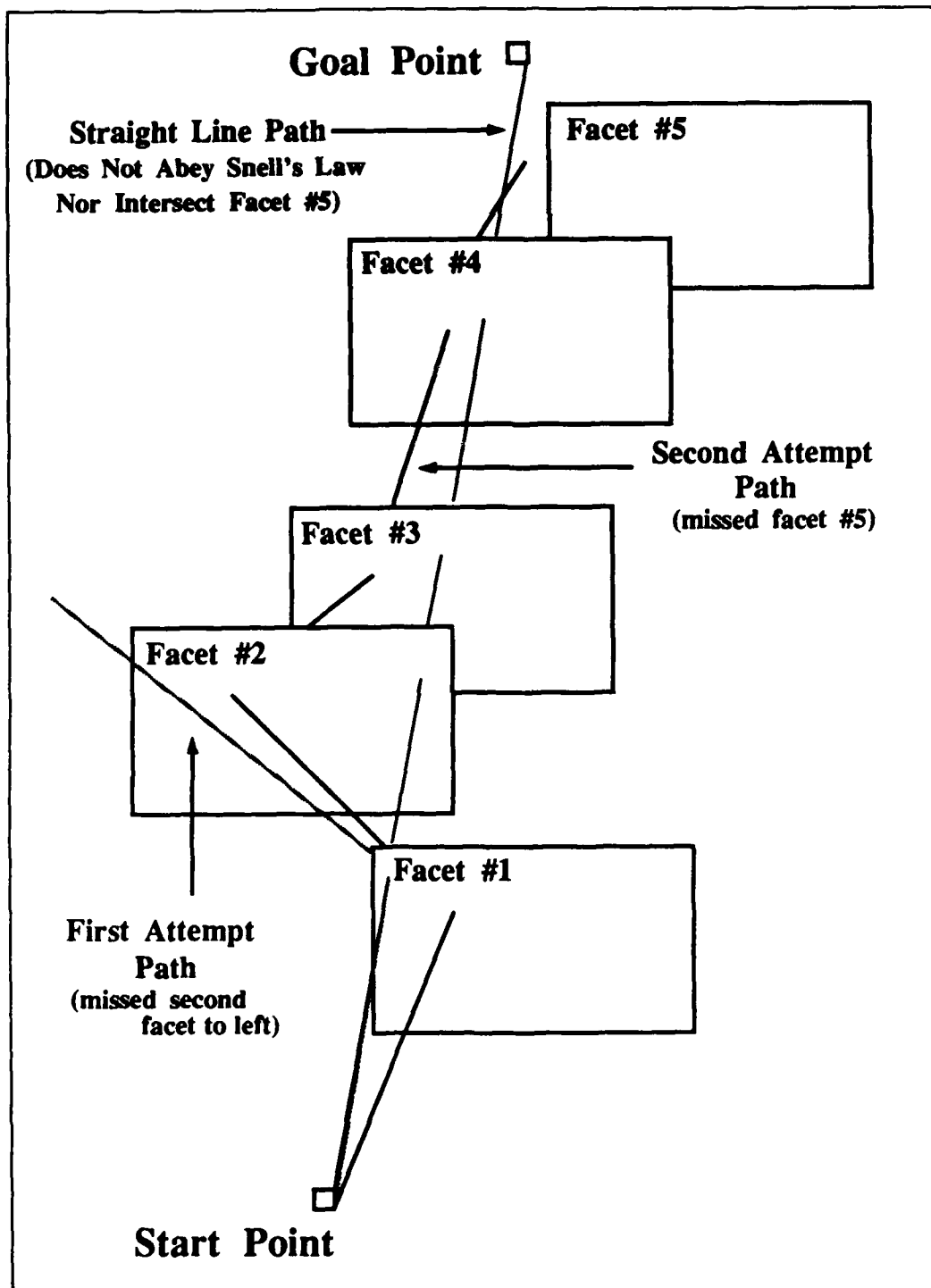
Figure 4-6. First Ray Adjustment
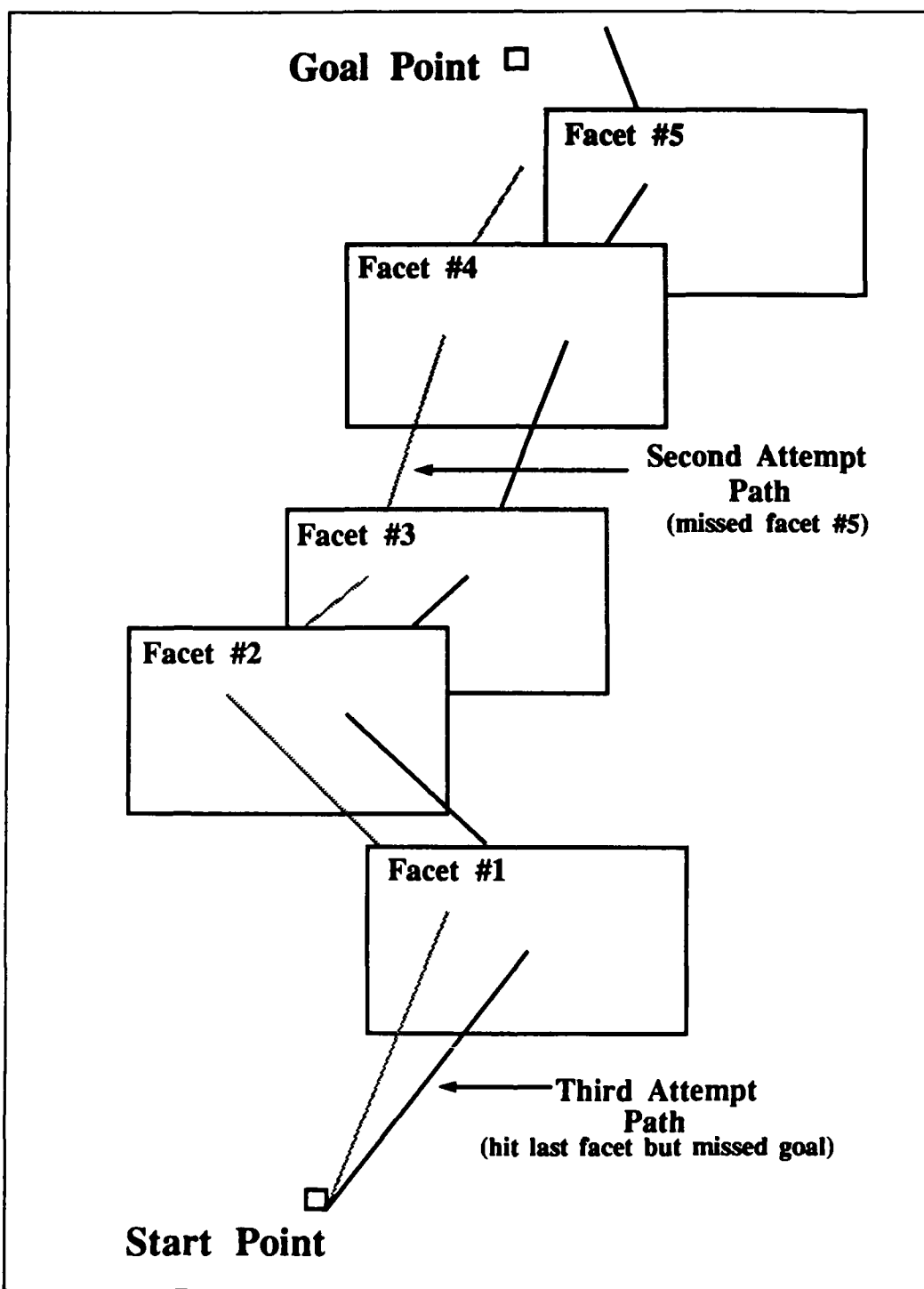
Figure 4-7. Ray Adjustment Into Final Volume
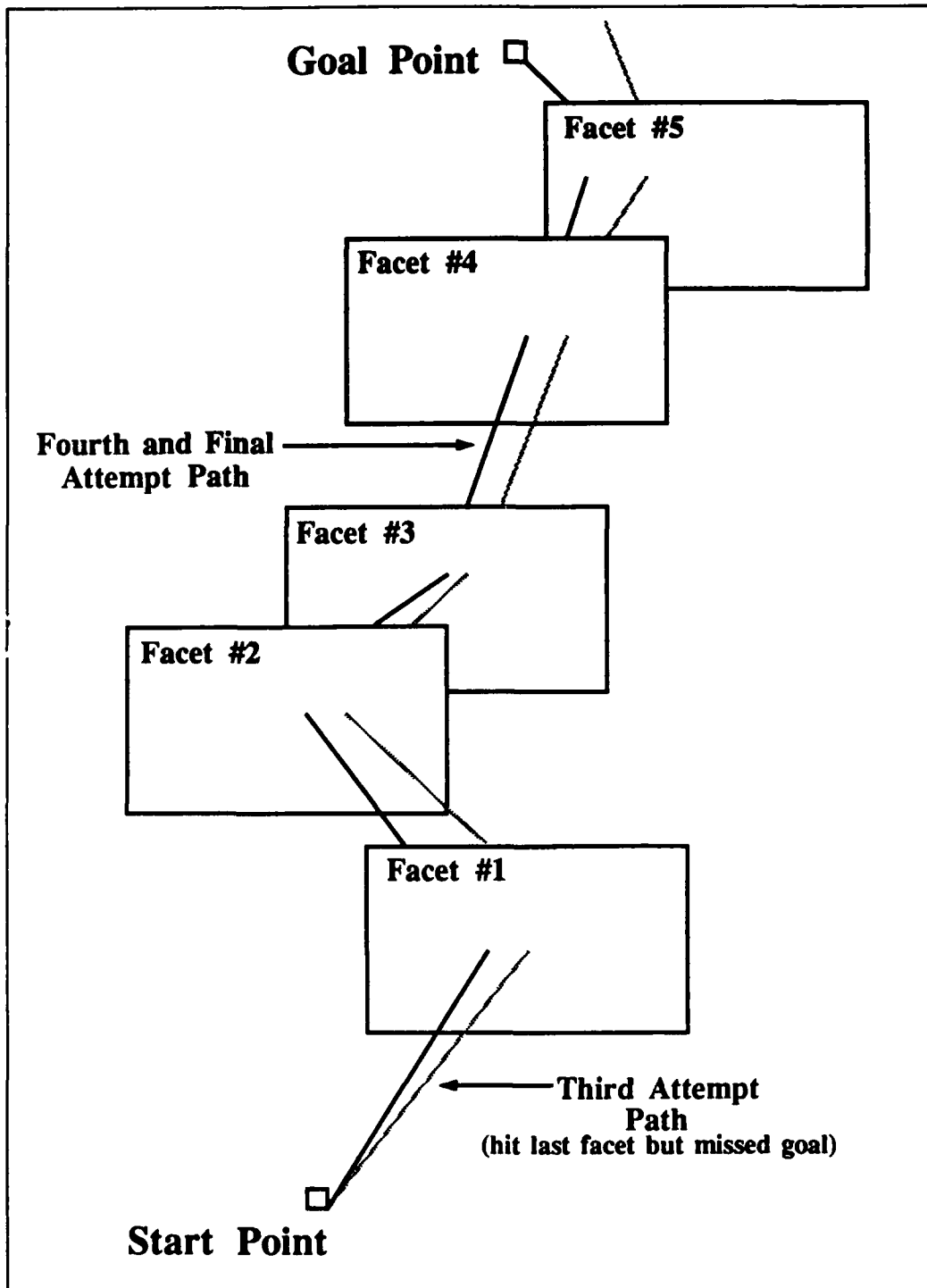
Figure 4-8. Final Random-Ray Adjustment Onto Goal Point

31

One problem encountered during the adjustment through the facets is depicted in Figure 4-9. The problem arises when passing from a high to a low probability-of-detection where a reflection can occur. Table 4-2 shows the allowable angle deviation from the normal to the facet that an inbound line to a facet can have and still pass through the facet according to Snell's Law. If an adjustment of the inbound ray puts its angle deviation from the normal outside this tolerance, our adjustment algorithm will not work. A different approach was taken so that once the facet had been intersected,

TABLE 4-2. Tolerance to Avoid Reflection

First Volume PD	Second Volume PD	Maximum Angle off Facet (rad)	Maximum Angle off Facet (deg)
0.010	0.010	1.5708	90.0000
0.015	0.010	0.7297	41.8103
0.020	0.010	0.5236	30.0000
0.025	0.010	0.4115	23.5782
0.030	0.010	0.3398	19.4712
0.035	0.010	0.2898	16.6016
0.040	0.010	0.2527	14.4775
0.045	0.010	0.2241	12.8396
0.050	0.010	0.2014	11.5370
0.055	0.010	0.1828	10.4757
0.060	0.010	0.1674	9.5941
0.065	0.010	0.1545	8.8499
0.070	0.010	0.1433	8.2132
0.075	0.010	0.1337	7.6623
0.080	0.010	0.1253	7.1808
0.085	0.010	0.1179	6.7563
0.090	0.010	0.1113	6.3794
0.095	0.010	0.1055	6.0423

but a reflection resulted, we change the adjustment vector to adjust to the projection of the last turn point on the facet.

When the random ray that hits the goal point has been found, it is passed with the original path to the REVISE-PATH module, to fill in details of the new path starting with this random-ray. This process is completed by making the Snell's Law adjustment at each successive facet until the goal is reached. The points of
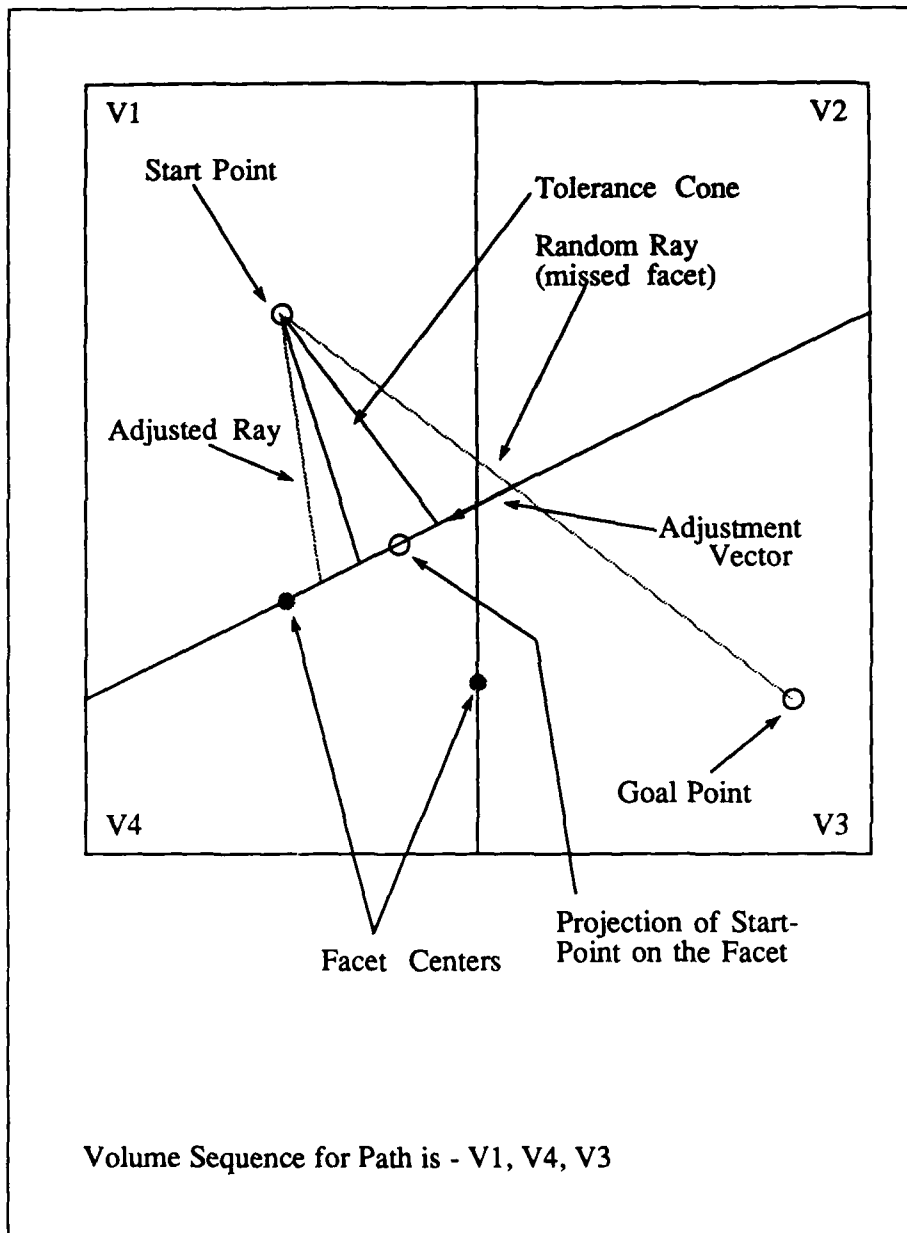
Figure 4-9. Over-Shoot Corrections

intersection of lines and facets are used to construct the new lines to the goal and both replace the old lines and points in the original path.

C. DISPLAY

1. System Requirements

The system used to implement a display was the Silicon Graphics IRIS 4D/70GT with eight megabytes of memory. The features of this machine such as drawing routines implemented in hardware, hidden-line removal, and lighting and shading routines made it an ideal choice. The machine is UNIX-based with the program written in C.

2. Input Files

The program reads in two files with the first being the terrain data (as in Table 4-3), and the second being a concatenation of all the paths you wish to display (as in Table 4-4). The paths must include the probability of detection along each segment.

3. Program Display

The purpose of this part of the program was to visualize the terrain and the paths created. The full screen is used to display the terrain and the paths drawn over it (Figure 4-10). The ground is drawn as a series of polygons with the variance in color produced by the lighting built into the IRIS. This reflected light is a function of the angle between the polygon's normal vector and the light source using Lambert's Cosine Law [Ref. 18]. The paths are colored according to their

34

TABLE 4-4. Example Path Data

3			
4			
4			
8			
10.00	300.00	-0.00	0.70
469.23	601.54	-300.00	0.70
420.00	668.00	-700.00	0.70
990.00	990.00	-990.00	0.70
10.00	300.00	-0.00	0.70
306.97	509.09	-300.00	0.70
702.92	787.87	-700.00	0.70
990.00	990.00	-990.00	0.70
10.00	300.00	-0.00	0.70
38.25	297.16	-81.26	0.70
175.07	303.45	-300.00	0.70
261.58	307.29	-437.44	0.70
300.05	309.00	-498.56	0.00
331.59	391.86	-700.00	0.00
390.88	438.18	-700.00	0.00
990.00	990.00	-990.00	0.70

TABLE 4-3. Example Terrain Data

7		
4		
0	300	-1000
0	300	0
350	500	-300
350	500	-1000
4		
1000	300	0
650	500	-300
350	500	-300
0	300	0
4		
650	500	-1000
650	500	-300
1000	300	0
1000	300	-1000
4		
350	500	-300
475	300	-400
475	300	-1000
350	500	-1000
4		
350	500	-300
650	500	-300
525	300	-400
475	300	-400
4		
525	300	-400
650	500	-300
650	500	-1000
525	300	-1000
4		
475	300	-400
525	300	-400
525	300	-1000
475	300	-1000

probability of detection along each line segment, varying from yellow to red as low to high probability. The dial controls, Figure 4-11, allow for rotation (Dial 0), tilt (Dial 1), and zoom (Dial 2) of the model. Additional controls including EXIT are provided by the mouse system.

4. Display Program Parts

The program can be broken into four main parts as shown in Figure 4-12. The first two sections, once completed, are never repeated. The second two sections are continuously updated and interact with each other to cause

Figure 4-10. Full Screen Display
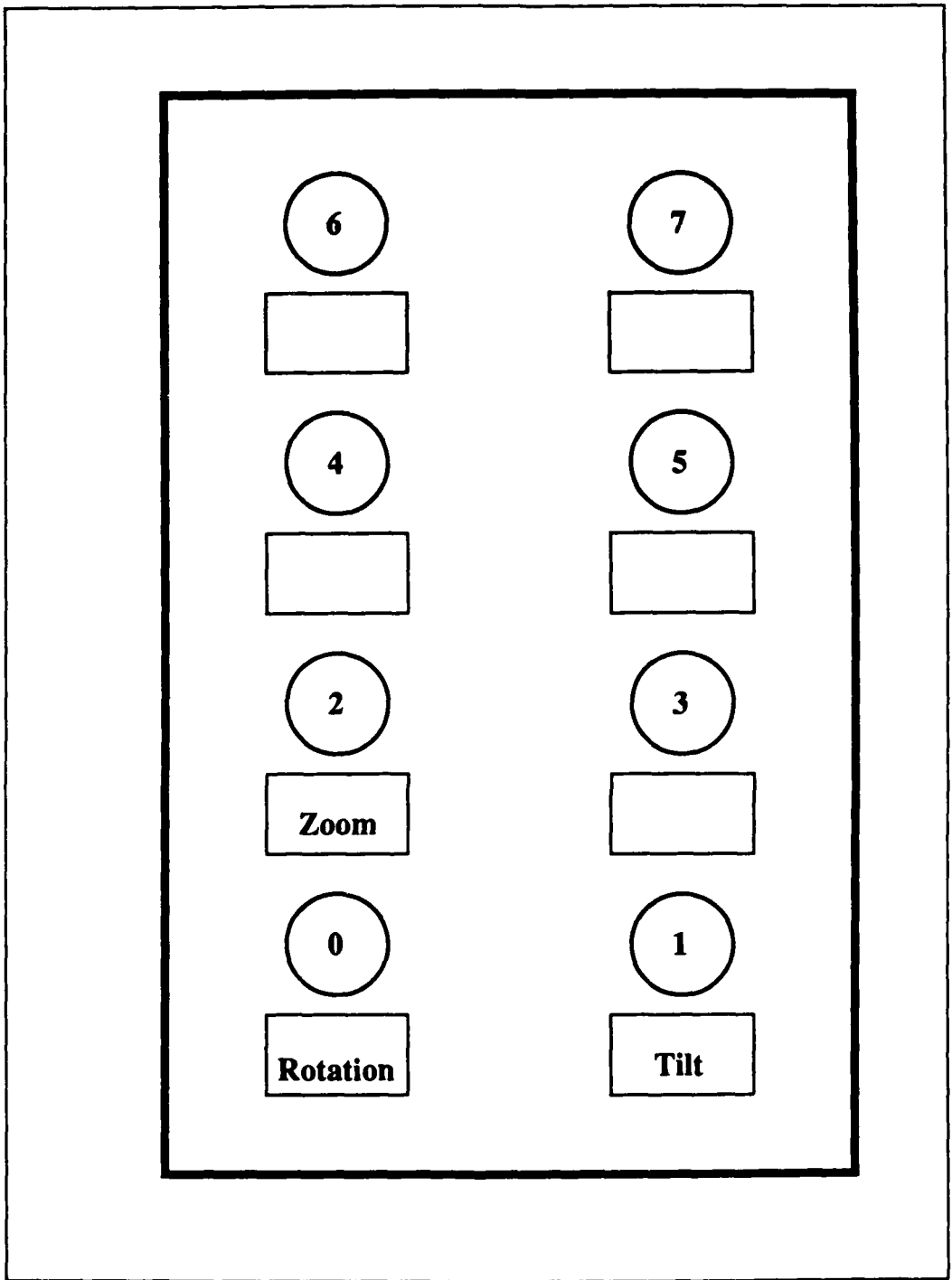
Figure 4-11. IRIS Control Box
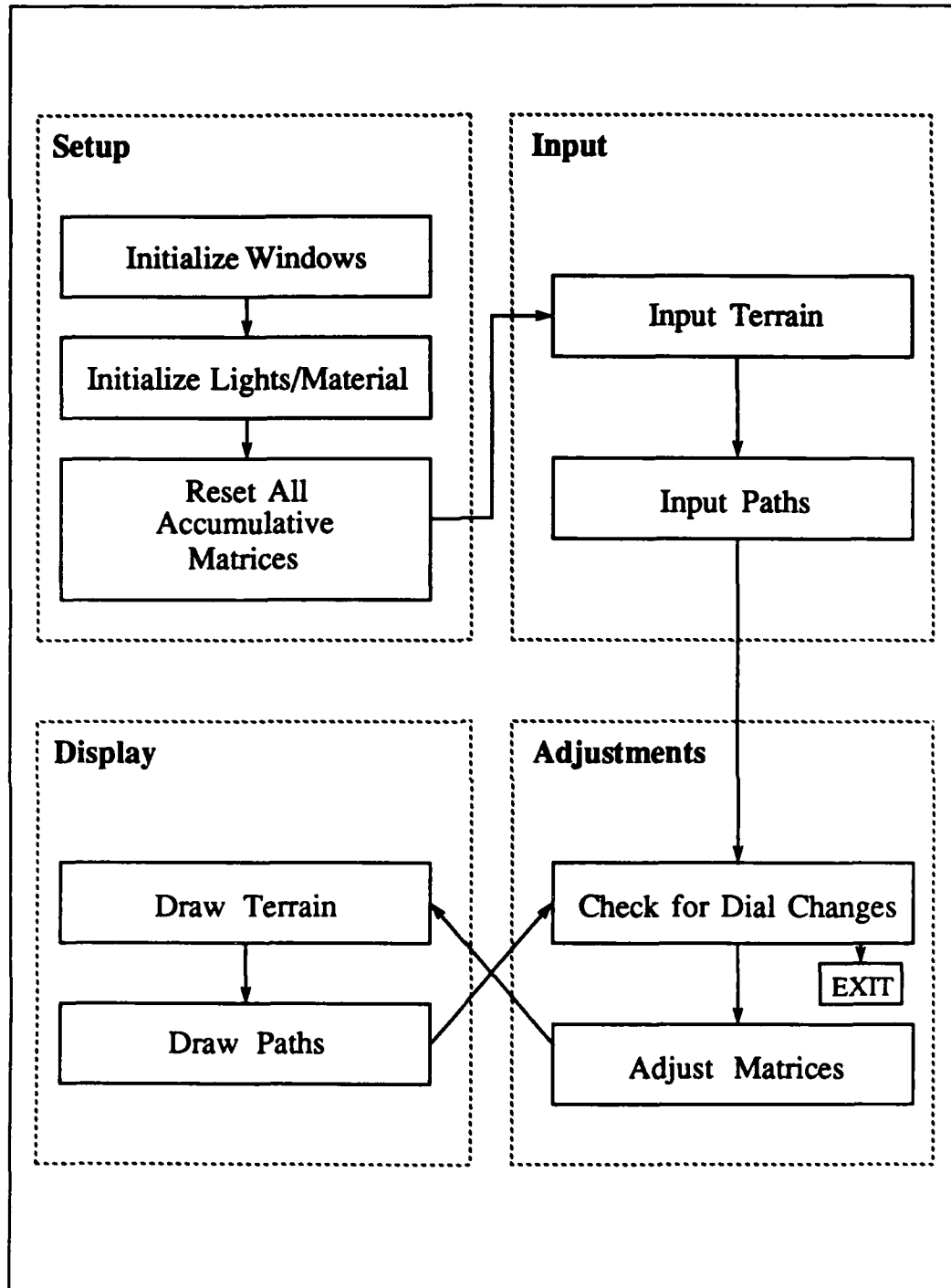
Figure 4-12. Block Diagram of Graphic Display Program

the changes in the display. This program was written for the course Computer Graphics, where many of the routines were given in class or in examples.

5. Data Structures

Two separate data structures was used for the terrain and the paths. The terrain is read in as a series of polygons defined by their three-dimensional vertices. The paths are read in as a series of three dimensional points with a probability of detection associated with each. These items are stored in array form and are adjusted and displayed each time the viewing angle or position changes.

6. Program Components

a. System Setup

All of the following are initialized: global variables, the IRIS window system, material and lighting models, and dial and menu controls. This allows the window system to be opened and cleared, and sets all the colors for the polygons and properties of the lighting models. Movement of the objects is facilitated by the use of accumulative matrices, so these are initially set to a unit matrix.

b. File Input

The two files are read in and processed one at a time. The terrain file is read in two parts: the base, which is read for all terrain, and the ground, which is unique to each area. As each is completed, the normal vectors are

computed and the colors and lighting properties are assigned to each polygon. Once the terrain is complete, the paths are read in with no calculations required.

c. *Display Terrain*

The initial data is displayed as it was input, with all the colors and lighting adjusted. At this point no other inputs have been received so the accumulative matrices are still in unit form and do not effect the terrain displayed. Subsequent displays will be altered by the matrices as adjusted by dial inputs.

d. *Control Inputs*

The inputs from the three dials are read and queued for alteration of the accumulative matrices. Dial zero allows you to rotate the terrain display left and right as shown in Figure 4-13. This rotation is about the center vertical axis (Y on the IRIS and Z on the TI Explorer). Each rotation is from the last displayed position and is not dependent on the dial's actual position. In other words, you can continuously rotate in one direction without reaching a stopping point. Dial one changes your eye position from ground level to a position directly above the terrain. This dial does have limitations as shown in Figure 4-13. The last input is zoom, on dial two. This increases or decreases the size of the picture. With this you must be careful because you can be looking at the terrain from inside of it, and this can be confusing.
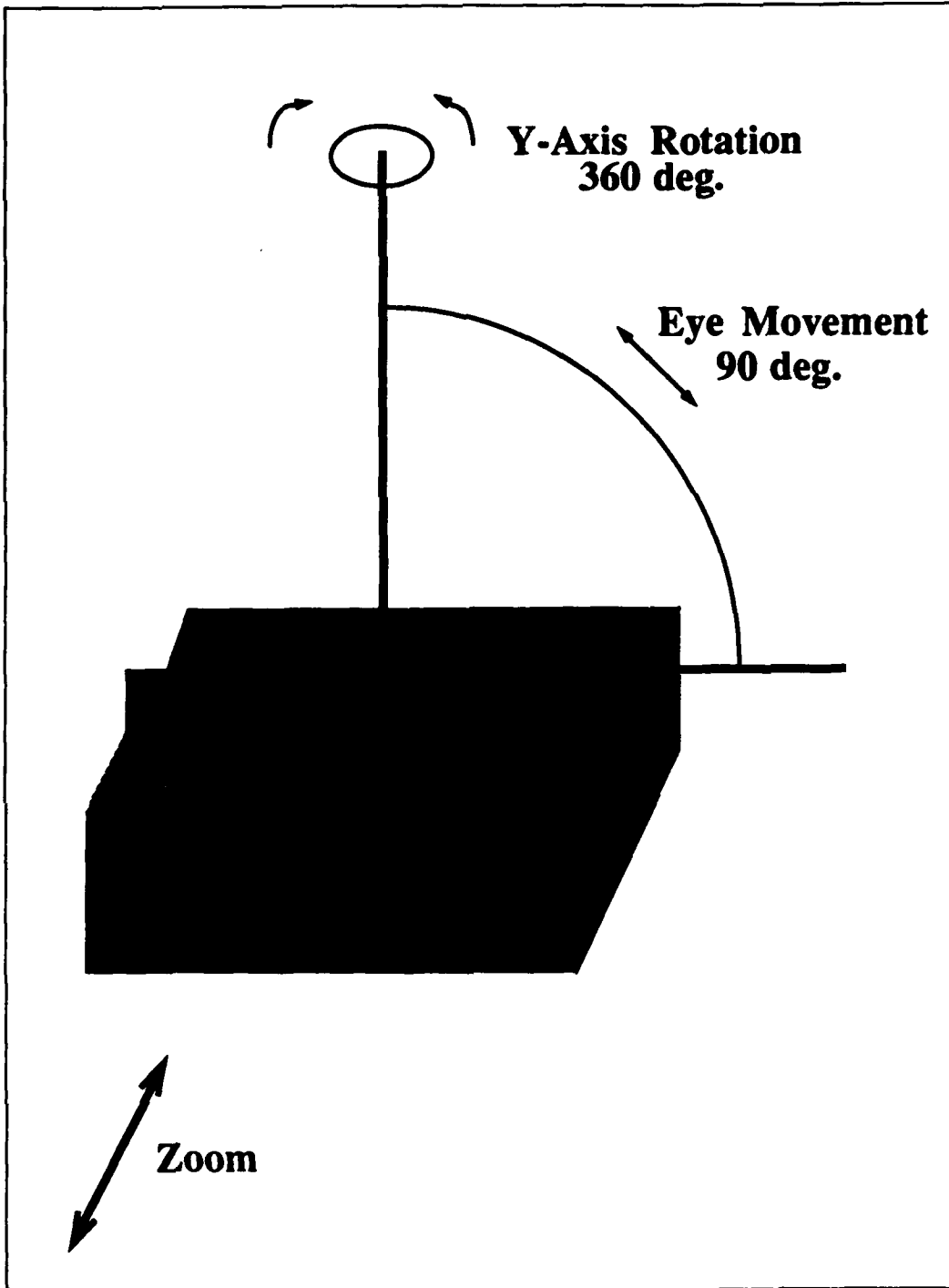
Figure 4-13. Graphic Display Limits

41

V. RESULTS

A. PATH PLANNING

1. Aircraft Realism

The A* search at this point does make its decisions based on true aircraft aerodynamics. The cost function relies heavily on the amount of fuel burned and how the missile will react to a path that will climb for a great deal of time. The processing of this data did not significantly increase the processing time in the [REF. 1] A* search. It is important to note that contemporary cruise missiles can store only a limited number of turn points, so if these are kept to a minimum, the better off we are.

2. Resultant Paths

Table 5-1 shows that random-ray optimization does indeed produce a more direct path from start to goal. Table 5-2 shows time required to obtain optimized paths with variable number of volumes, and Table 5-3 shows how many single optimizations had to be run on a path to obtain the same results (within limits) as the random-ray optimization.

42

Table 5-1. Random-Ray Paths vs Original Paths

Terrain	Old-Path	Distance	Time	New-Path	Distance	Time
T-27	0003	999.7	133.3	0008	599.9	80.0
T-21	0019	894.1	119.1	0020	815.0	108.7
T-27	0004	1481.9	198.1	0009	1012.2	135.0
T-27	0023	1591.1	212.1	0052	1393.0	185.7

TABLE 5-2. Run-Time to obtain Random-Ray Paths vs Old Optimized Paths

Terrain	No of Volumes In Path	Run Time(sec) Random-Ray	Run Time(sec) Old-Optimize
T-27	5	107	8
T-21	5	239	42
T-25	11	270	*
T-21	5	165	40
T-27	4	7	4

* - Would not optimize

Table 5-3. Comparison of Old Optimize vs Random-Ray

Terrain	Random-Ray Path Cost	Number of Optimizations	Cost After Old-Optimize
T-27	881.4	3	981.5
T-21	647.2	6	683.7
T-25	983.7	3*	*
T-21	659.1	6	675.2
T-27	1303.2	3	1303.3

* - Started to diverge after 3rd optimization run

Figure 5-1 through 5-3 show the original path and the optimized path (the optimized path is the straighter of the two) with Table 5-4 through 5-6 giving the corresponding data on each. Figure 5-4 and Table 5-7 show that the results obtained by the computer can be improved on but not by much.
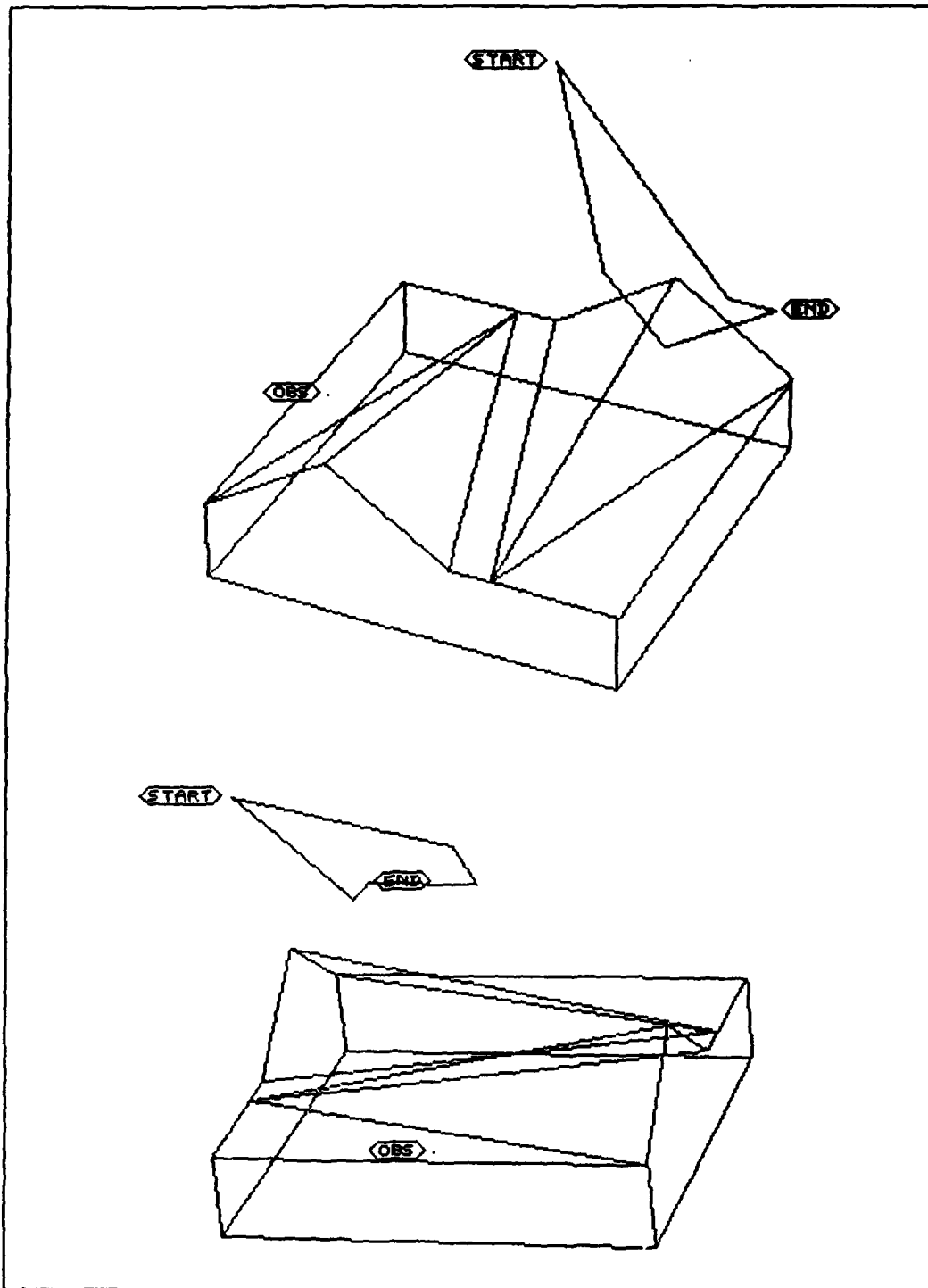
Figure 5-1. Original vs Random-Ray Optimized Paths

TABLE 5-4. Original and Random-Ray Path Data

```
> (path-data '|path0003|)
            Leg   Total  Leg    Total  Leg   Fuel    Vol    PD     Leg
Point       Time  Time   Dist   Dist   Fuel  Remain  PD     Cost   Cost
(10.0 400.0 910.0)
            0.0   0.0    0.0    0.0    0.0   1500    -      -      -
(420.0 700.0 668.0)
            67.7  67.7   508.0  508.0  337.6 1162.4  0.020  135.5  473.1
(420.0 852.5 505.0)
            20.3  88.1   152.5  660.5  100.9 1061.5  0.020  40.7   141.6
(110.0 990.0 450.0)
            45.2  133.3  339.1  999.7  225.8 835.6   0.000  0.0    225.8
Total cost of this path -   840.5
minimum PD cost -     0.0
maximum PD cost -   135.5
average PD cost -     6.3

840.5207
> (path-data '|path0008|)
            Leg   Total  Leg    Total  Leg   Fuel    Vol    PD     Leg
Point       Time  Time   Dist   Dist   Fuel  Remain  PD     Cost   Cost
(10.0 400.0 910.0)
            0.0   0.0    0.0    0.0    0.0   1500    -      -      -
(71.3271 700.0 622.3257)
            40.8  40.8   306.2  306.2  202.8 1297.2  0.020  81.7   284.5
(107.56799 877.03156 452.3262)
            24.1  64.9   180.7  486.9  119.7 1177.4  0.020  48.2   167.9
(110.0 990.0 450.0)
            15.1  80.0   113.0  599.9  75.3  1102.1  0.000  0.0    75.3
Total cost of this path -   527.7
minimum PD cost -     0.0
maximum PD cost -    81.7
average PD cost -     6.6

527.7141
> (dribble)
```

46
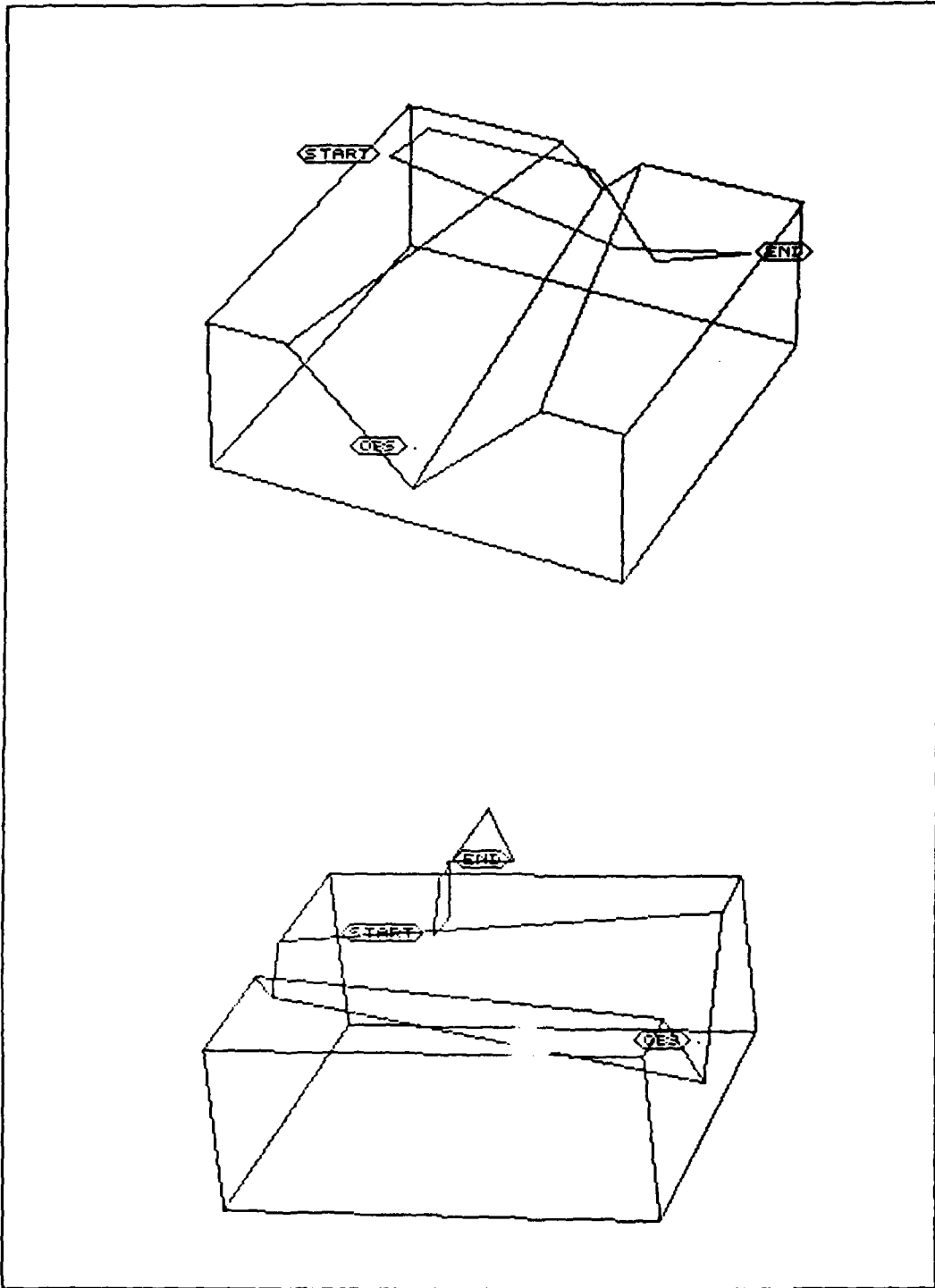
Figure 5-2. Original vs Random-R... Optimized Paths

47

TABLE 5-5. Original and Random-Ray Path Data

```
> (path-data '|path0019|)
              Leg   Total  Leg    Total  Leg    Fuel    Vol    PD     Leg
Point         Time  Time   Dist   Dist   Fuel   Remain  PD     Cost   Cost
(500.0 200.0 600.0)
              0.0   0.0    0.0    0.0    0.0    1500    -      -      -
(500.0 300.0 700.0)
              13.3  13.3   100.0  100.0  67.1   1432.9  0.015  20.0   87.1
(500.0 700.0 700.0)
              53.3  66.7   400.0  500.0  266.7  1166.2  0.015  80.0   346.7
(500.0 850.0 500.67114)
              20.0  86.7   150.0  650.0  99.1   1067.1  0.015  30.0   129.1
(300.0 990.0 440.0)
              32.6  119.2  244.1  894.1  162.5  904.6   0.000  0.0    162.5
Total cost of this path -    725.4
minimum PD cost -       0.0
maximum PD cost -      80.0
average PD cost -       6.1

725.37067
>
> (path-data '|path0020|)
              Leg   Total  Leg    Total  Leg    Fuel    Vol    PD     Leg
Point         Time  Time   Dist   Dist   Fuel   Remain  PD     Cost   Cost
(500.0 200.0 600.0)
              0.0   0.0    0.0    0.0    0.0    1500    -      -      -
(474.61185 305.07764 563.2685)
              14.4  14.4   108.1  108.1  71.9   1428.1  0.015  21.6   93.5
(384.7632 676.95264 433.27493)
              51.0  65.4   382.6  490.7  254.5  1173.6  0.015  76.5   331.0
(377.15268 708.35095 422.09427)
              4.3   69.7   32.3   523.0  21.5   1152.1  0.015  6.5    27.9
(300.0 990.0 440.0)
              38.9  108.7  292.0  815.0  194.8  957.4   0.000  0.0    194.8
Total cost of this path -    647.2
minimum PD cost -       0.0
maximum PD cost -      76.5
average PD cost -       6.0

647.21844
> (dribble)
```

48
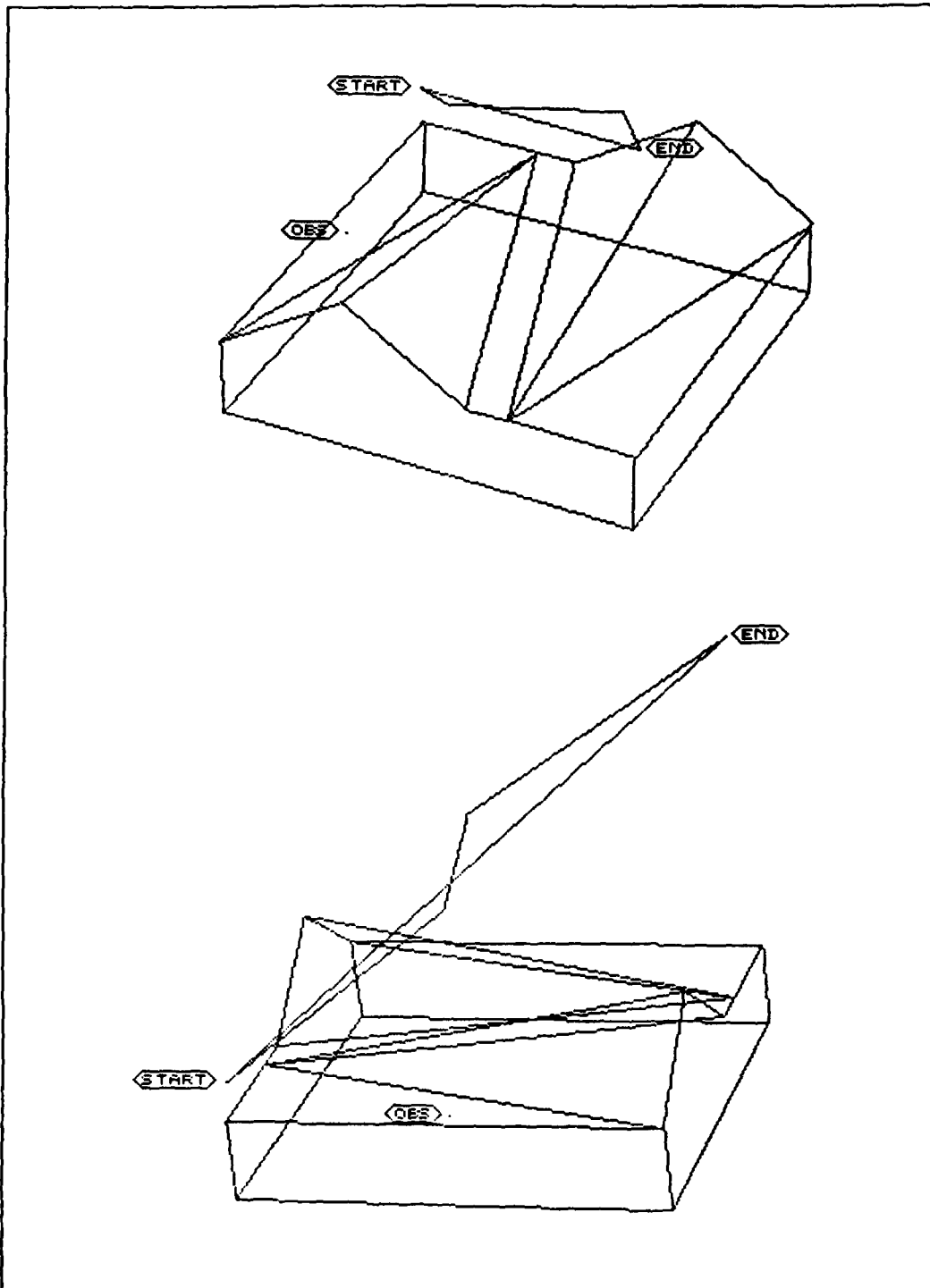
Figure 5-3. Original vs Random-Ray Optimized Paths

49

TABLE 5-6. Original and Random-Ray Path Data

```
> (path-data ' |path0023|)
             Leg    Total   Leg    Total   Leg    Fuel    Vol     PD      Leg
Point        Time   Time    Dist   Dist    Fuel   Remain  PD      Cost    Cost
(10.0 0.0 300.0)
             0.0    0.0     0.0    0.0     0.0    1500    -       -       -
(469.23077 300.0 601.53845)
             73.1   73.1    548.5  548.5   367.0  1133.0  0.020   146.3   513.3
(420.0 700.0 668.0)
             53.7   126.9   403.0  951.6   269.0  864.0   0.020   107.5   376.4
(990.0 990.0 990.0)
             85.3   212.1   639.5  1591.1  427.8  436.2   0.020   170.5   598.3
Total cost of this path -  1488.1
minimum PD cost -    107.5
maximum PD cost -    170.5
average PD cost -      7.0

1488.1104
>
>
> (path-data ' |path0052|)
             Leg    Total   Leg    Total   Leg    Fuel    Vol     PD      Leg
Point        Time   Time    Dist   Dist    Fuel   Remain  PD      Cost    Cost
(10.0 0.0 300.0)
             0.0    0.0     0.0    0.0     0.0    1500    -       -       -
(306.9697 300.0 509.0909)
             56.3   56.3    422.1  422.1   282.4  1217.6  0.020   112.6   394.9
(702.92303 700.0 787.8744)
             75.0   131.3   562.8  985.0   376.5  841.2   0.020   150.1   526.6
(990.0 990.0 990.0)
             54.4   185.7   408.1  1393.0  272.9  568.2   0.020   108.8   381.8
Total cost of this path -  1303.2
minimum PD cost -    108.8
maximum PD cost -    150.1
average PD cost -      7.0

1303.2467
>
>
> (dribble)
```
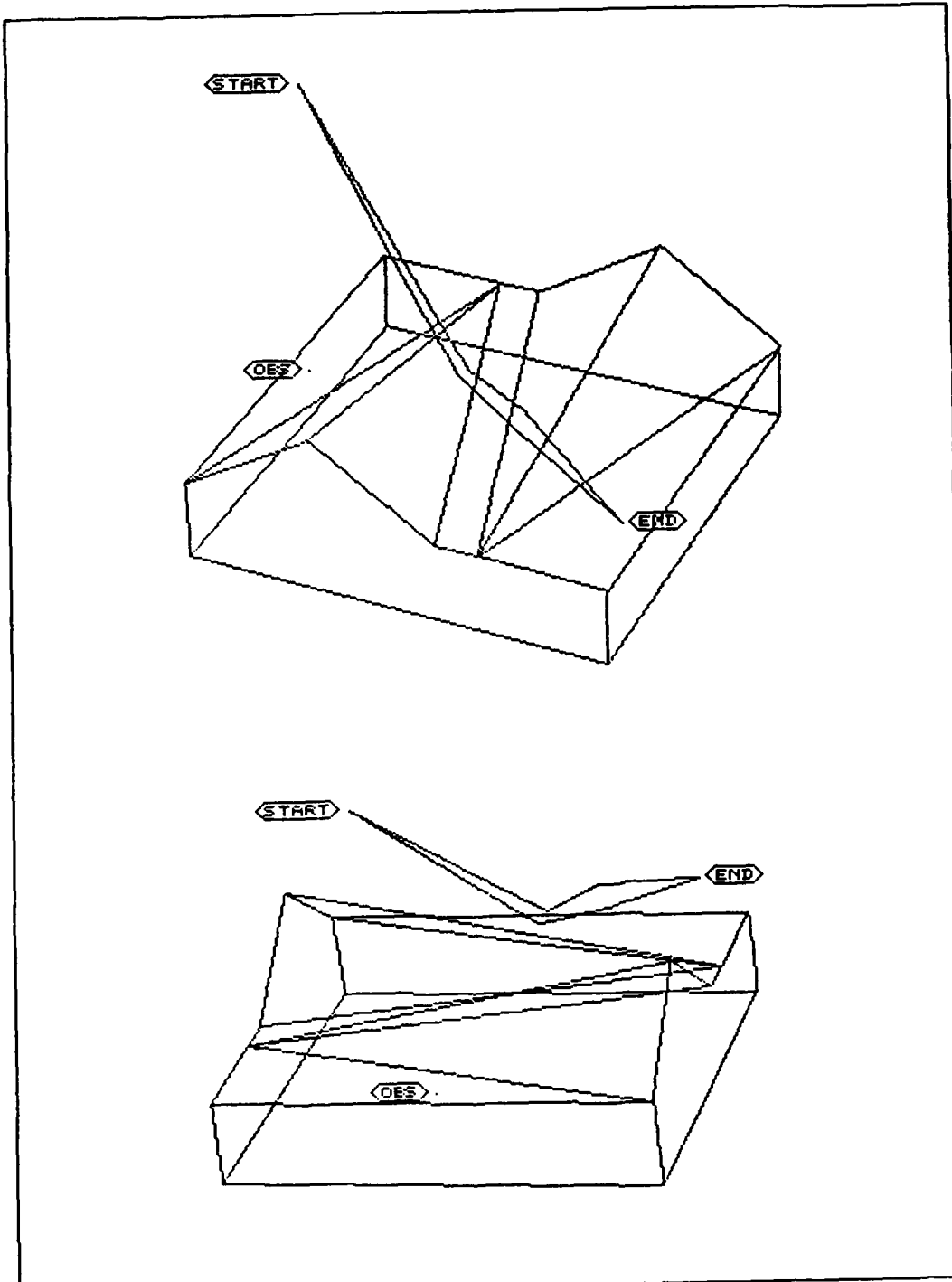
Figure 5-4. Computer Optimized vs User Optimized

TABLE 5-7. Computer Optimized vs User Optimized

```
> (path-data '|path0029|)
             Leg    Total   Leg    Total  Leg    Fuel    Vol    PD      Leg
Point        Time   Time    Dist   Dist   Fuel   Remain  PD     Cost    Cost
(410.0 10.0 900.0)
             0.0    0.0     0.0    0.0    0.0    1500    -      -       -
(550.05664 300.0 633.2163)
             42.9   42.9    322.0  322.0  213.5  1286.5  0.020  85.9    299.4
(660.2497 528.16956 423.31775)
             33.8   76.7    253.4  575.4  168.0  1118.5  0.020  67.6    235.6
(741.65796 700.0 411.26544)
             25.4   102.1   190.1  765.6  126.7  991.8   0.000  0.0     126.7
(900.0 990.0 300.0)
             44.1   146.1   330.4  1096.0 219.8  772.0   0.000  0.0     219.8
Total cost of this path -    881.4
minimum PD cost -     0.0
maximum PD cost -     85.9
average PD cost -      6.0


881.4159
> (path-data '|path0037|)
             Leg    Total   Leg    Total  Leg    Fuel    Vol    PD      Leg
Point        Time   Time    Dist   Dist   Fuel   Remain  PD     Cost    Cost
(410.0 10.0 900.0)
             0.0    0.0     0.0    0.0    0.0    1500    -      -       -
(556.0 300.0 601.0)
             43.3   43.3    324.7  324.7  215.1  1284.9  0.020  86.6    301.7
(650.91486 491.0918 406.6197)
             28.4   71.7    213.4  538.0  141.4  1143.5  0.020  56.9    198.3
(754.3238 700.0 353.66742)
             31.1   102.8   233.1  771.1  155.2  988.4   0.000  0.0     155.2
(900.0 990.0 300.0)
             43.3   146.1   324.5  1095.7 216.1  772.2   0.000  0.0     216.1
Total cost of this path -    871.2
minimum PD cost -     0.0
maximum PD cost -     86.6
average PD cost -      6.0


871.23987
> (dribble)
```

B.   DISPLAY

1.   Terrain Models

Because transparent three-dimensional line drawings of terrain are acceptable only to the trained eye, a display that uses solid figures and hidden line removal is much preferred. Figure 5-5 through 5-7 show the contrast in understandability of the line drawings versus graphic depiction. As the models become even more complex the need for better displays increases. Figure 5-8 through 5-10 show how adding multiple paths has little effect on the readability.
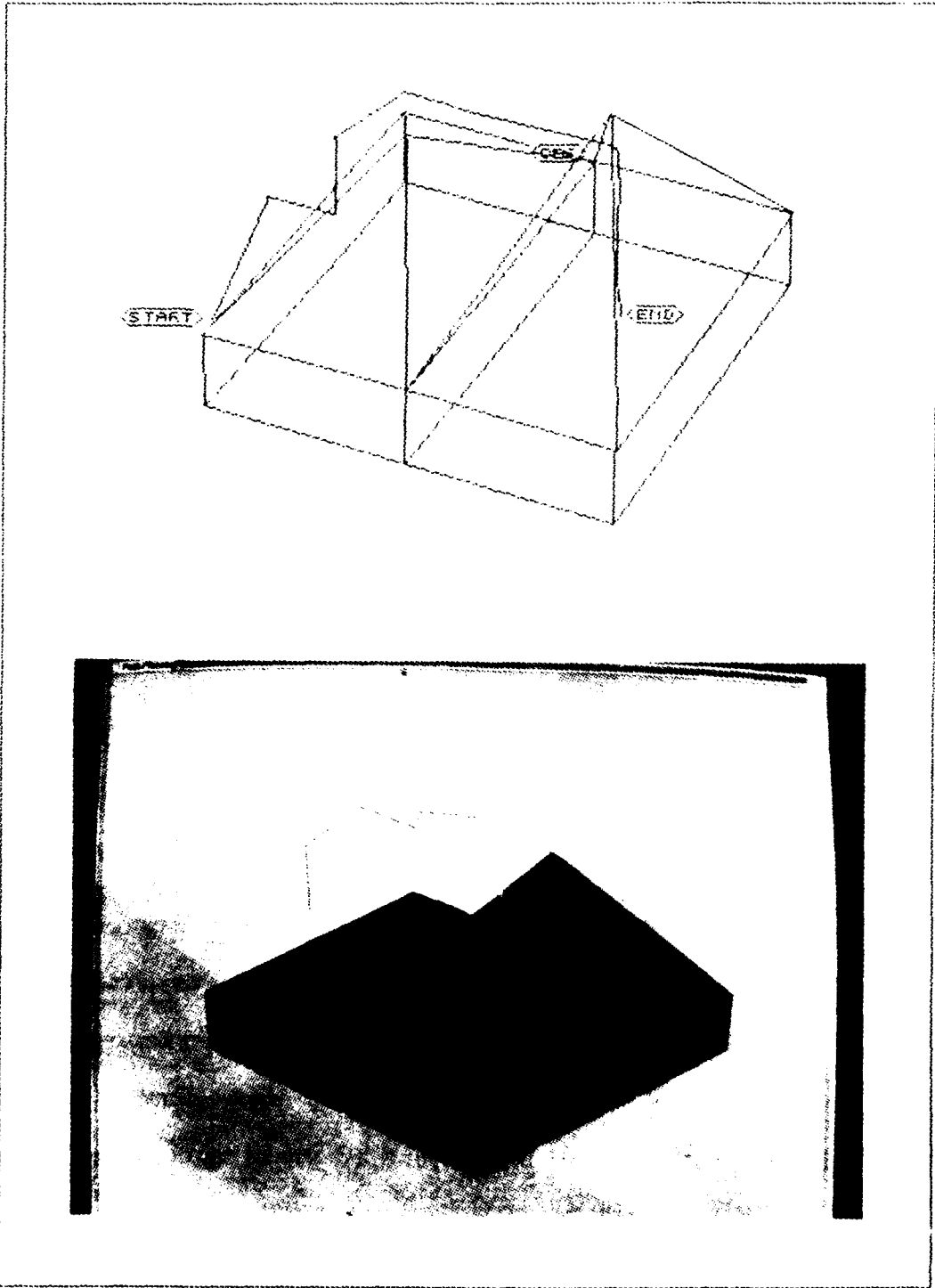
Figure 5-5. Line Drawing vs Graphic Display
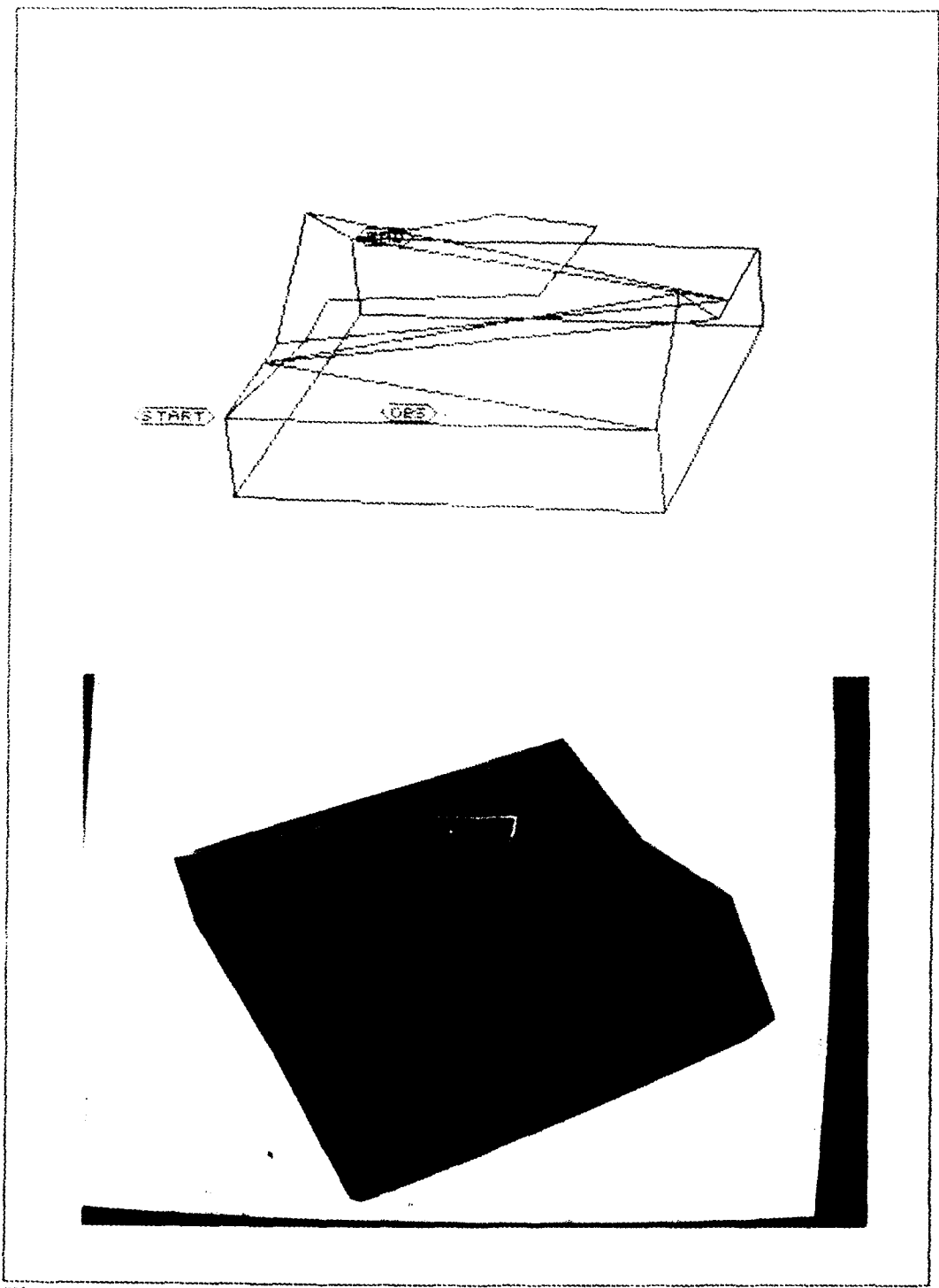
Figure 5-6. Line Drawing vs Graphic Display
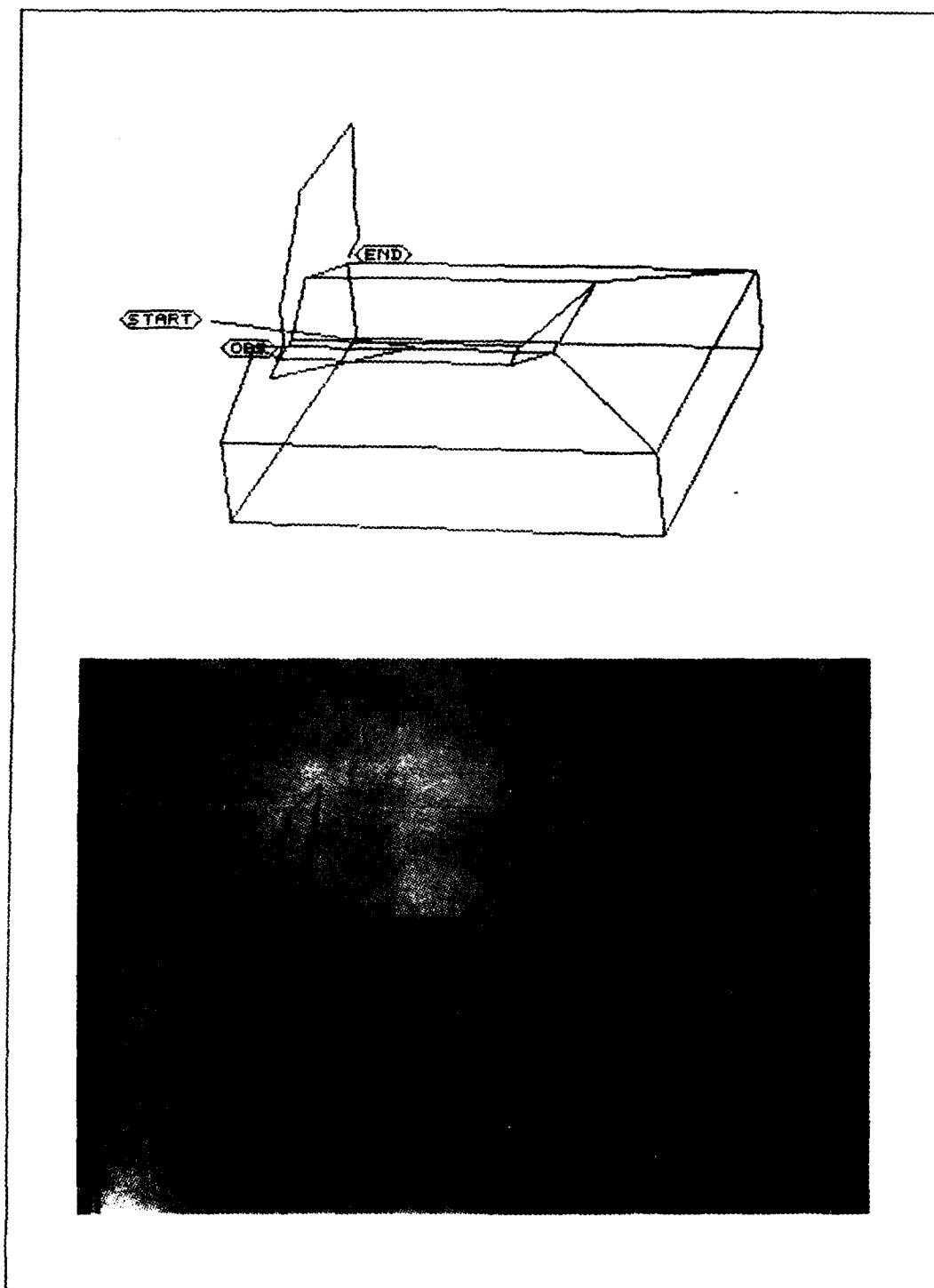
Figure 5-7. Line Drawing vs Graphic Display

56
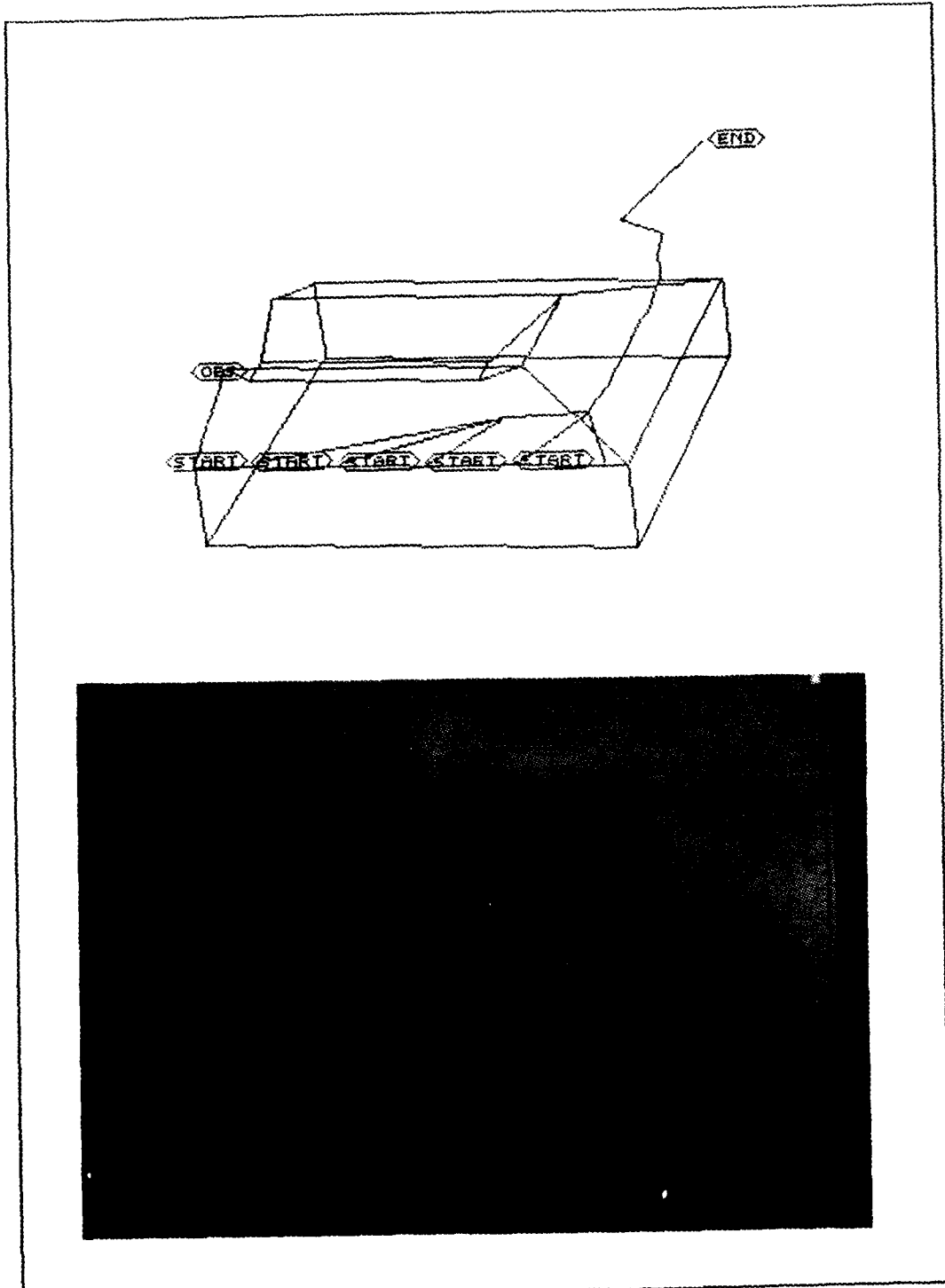
Figure 5-8. Line Drawing vs Graphic Display (Multi-Path)
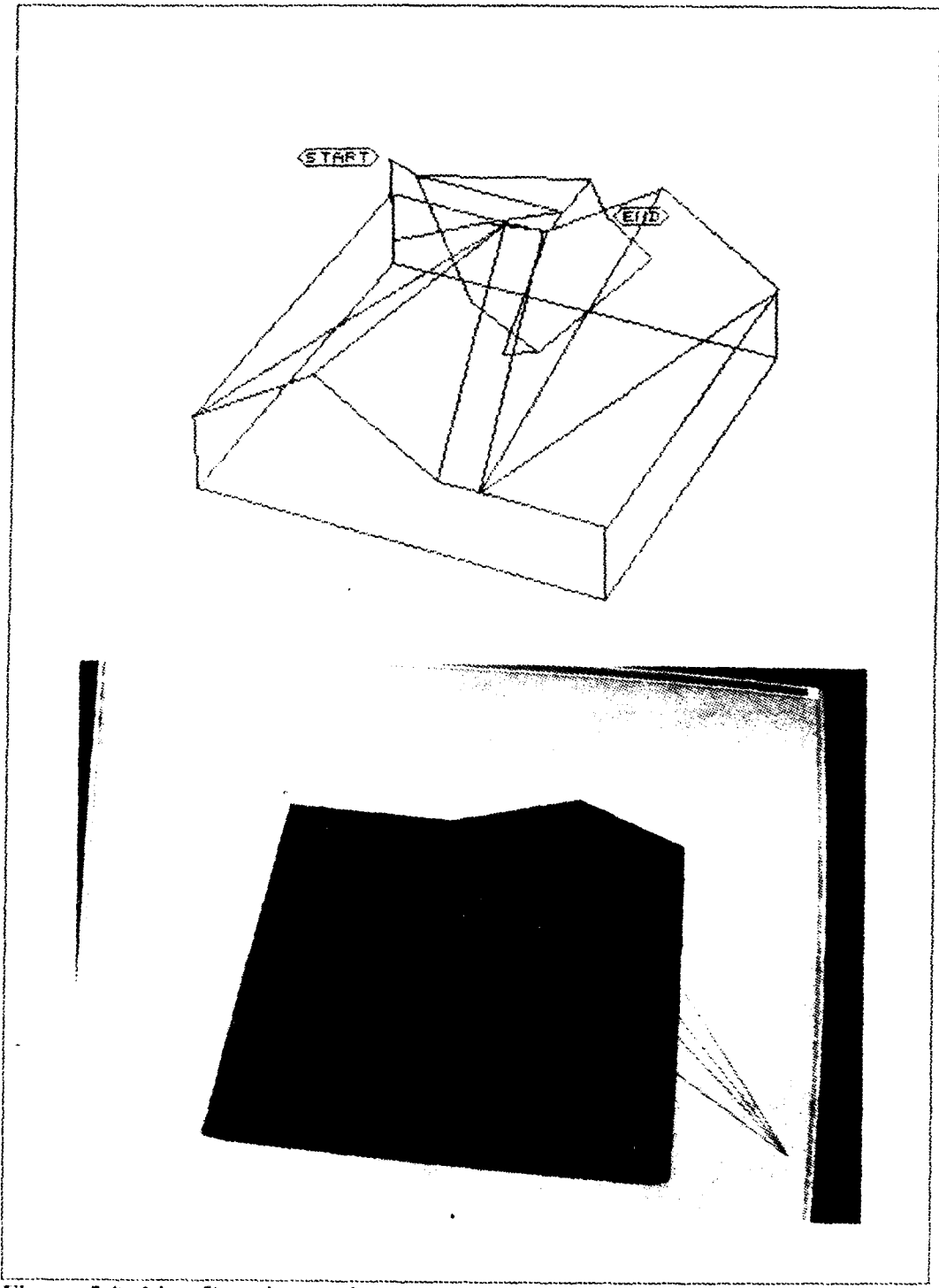
Figure 5-9. Line Drawing vs Graphic Display (Multi-Path)
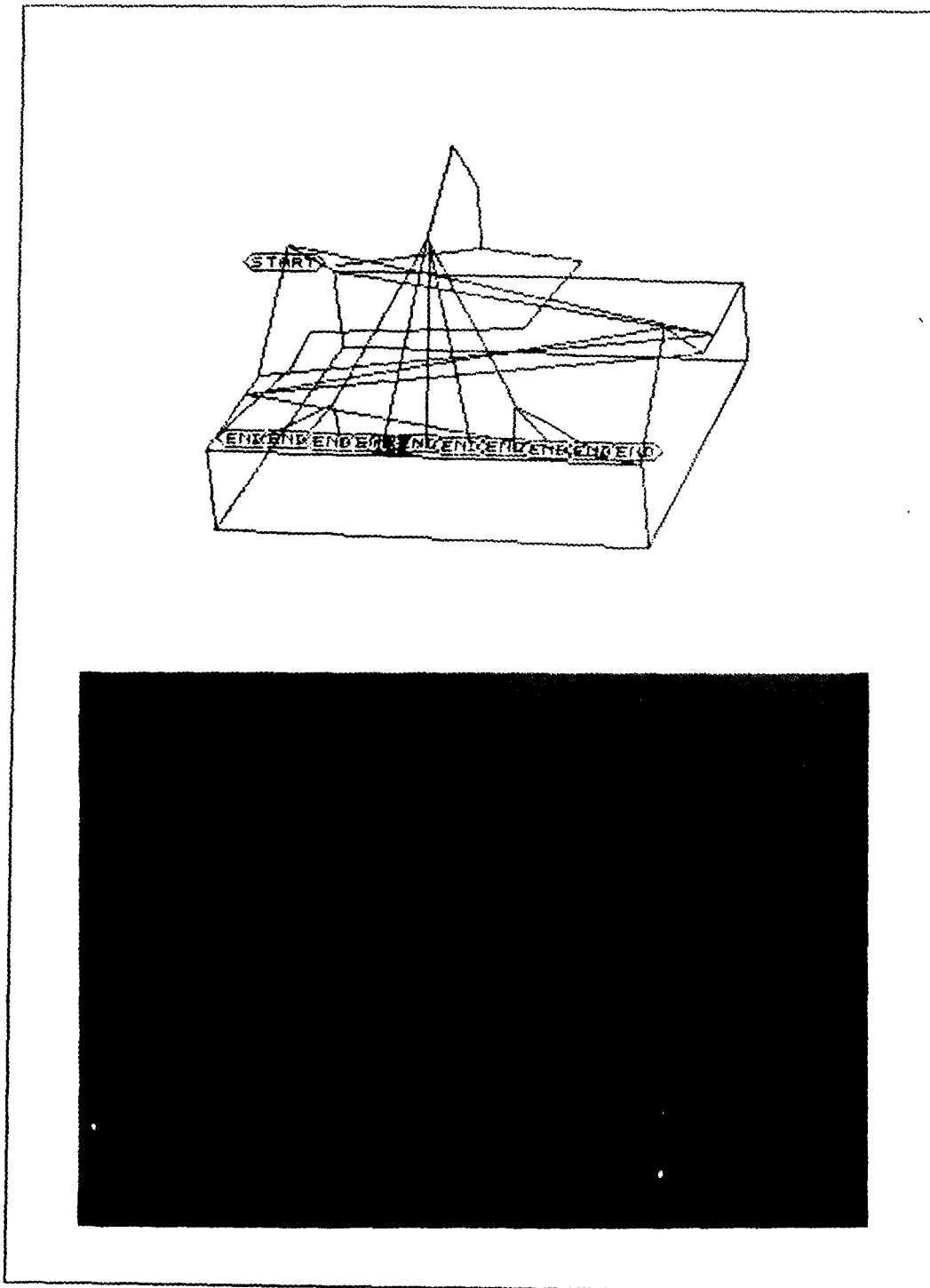
Figure 5-10. Line Drawing vs Graphic Display (Multi-Path)

2. Viewpoint and Perspective

Using the graphics display we are able to place ourself at the location of the observer and see what he might see (Figure 5-11) or view the path from any angle, as shown previously. This is a great advantage when making the final decision on which path is best.
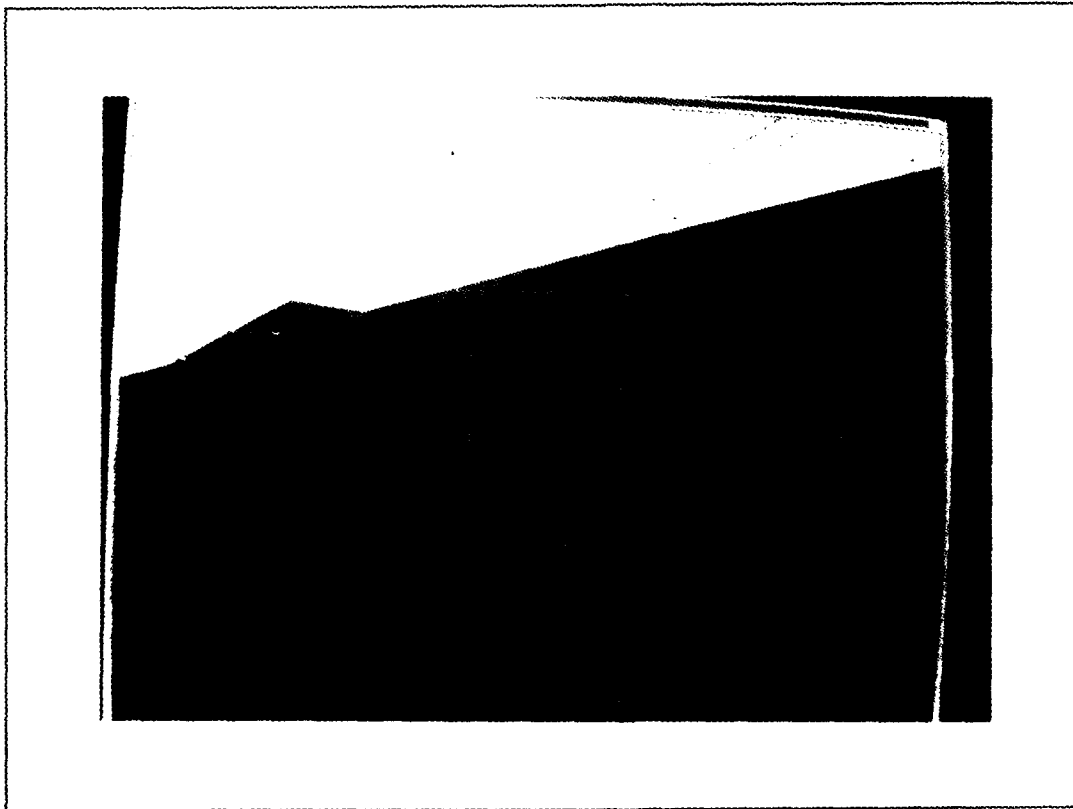
Figure 5-11. View From Observer Position

VI. CONCLUSIONS

A. DISCUSSION

All goals set at the beginning of this project have been met: to modify [Ref. 1] to include a more aerodynamic model, to graphically display our model, and to implement a new optimization technique. The flight characteristics of most Navy aircraft can be modeled and encoded in the aircraft section of the program. This data can be extracted from the various NATOPS manuals for the aircraft or obtained from Naval Labs.

The graphic display was developed to confirm that a three-dimensional display was possible and was useful in showing the optimality of our best path. The jetcard type printouts are helpful for the actual figures such as time and fuel used, but to get a full feeling of the path, the graphic display is a must.

The advantage of our method of optimization is the speed at which a solution can be obtained. As shown in Chapter V, the random-ray method eliminates the vast majority of paths to explore and optimize. The optimization is obtained in one pass so no further calculations are required to see if a better path exists. If no random-ray optimization path exists, we are no worst off then we were to begin with, so we can use the approach of [Ref. 1] to optimize each path individually.

B. KNOWN PROBLEMS

There is no treatment of the paths around obstacles by the random-ray technique. Choices need to be made as to how to detour around objects. Presently, if Snell's Law cannot bend the path to avoid the object, it says no path is available.

The observers we have modeled have unlimited line-of-sight capabilities, not affected by the range limitations. Such details can be added to the program as well as adjustments for diffusion, diffraction, and refraction.

No attempt was made to allow for weighting of optimization criteria. The main criteria can remain minimal detection, but a choice can be made whether to maximize fuel utilization or time. Items such as aircraft speed, altitude and fuel load can also be weighted.

The graphic display runs separately and addresses only our limited models of terrain. The program needs to be expanded to allow altering the paths displayed while the program is running, and to display larger areas of terrain and real terrain data such as in [Ref. 8].

Elements that affect aircraft and aircrew performance have not been included. Items like temperature, winds and severe weather could be included as properties of each volume. Variations in altitude and speed, for changes in visibility conditions, terrain type (mountainous, hilly or flat), and aircrew ability need to be addressed.

As discussed in chapter IV, Snell's Law is very susceptible to reflection if the danger (probability-of-detection) varies much from volume to volume. This can be avoided by standardizing the values for probability-of-detection so that the maximum

is no more then 0.05. Note that a volume that is not visible is automatically given a value of 0.01 to avoid a division-by-zero error.

C. RECOMMENDATIONS

Execution time is going to be significant no matter what machine the program is implemented on. If we can store the results of the division of airspace into volumes, we can do only once the initial processing, the most time-intensive phase, and use the stored data from then on. Because of the way labels are generated by the TI Explorer for the objects we use (points, lines, facets, volumes, and so on,), we are limited to 9999 of each type. When random-ray optimization is run, many labels generated are not used more then once, which depletes the list after only a limited number of paths have been tested. This should be fixed.

Another optimization technique which can be implemented is to restrict the set of directions before the selection of the random ray. This was demonstrated in two dimensions by Ron Ross [Ref. 19]. To do this, find the range of all possible rays that will pass through the first window from the start point. As these rays pass through the window, apply Snell's Law and see which of these pass through the second window. The rays not passing through the second window can now be eliminated from the original set of directions. This can be continued until the goal is reached or until no rays pass through the next window.

63

LIST OF REFERENCES

1. Lewis, David H., *Optimal Three-Dimensional Path Planning Using Visibility Constraints*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

2. Naval Oceanography Command, *Aviators Guide to OPARS Flight Planning*, by John P. Garthner, June 1987.

3. Naval Postgraduate School Report NPS67-82-003, *HP-41CV Flight Performance Advisory System (FPAS) for the E-2C, E2-B, and C-2 Aircraft*, by Dennis R. Ferrell, June 1977.

4. Campbell, Richard W., and Champney, Robert K., *The A-6E/HP-41CV pocket Sized Flight Performance Advisory System*, unpublished, Naval Postgraduate School, Monterey, California, December 1981.

5. Hargrave, Douglas F., *Development of the A-6E/A-6E TRAM/KA-6D NATOPS Calculator Aided Performance Planning System (NCAPPS)*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1983.

6. Nutter, Christopher G., *Development of Flight Performance Algorithms and a Tactical Computer Aided Mission Planning System For The A-7E Aircraft*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1986.

7. Rowe, Neil C., *Artificial Intelligence Through Prolog*, Prentice-Hall, Inc., 1988.

8. Fichten, Mark A., and Jennings, David H., *Meaningful Real-Time Graphics Workstation Performance Measurements*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

9. Huisken, R., *The Origin of the Strategic Cruise Missile*, pp. 3-12, Praeger Publishers, 1981.

10. Tsipis, K., "Cruise Missiles," *Scientific America*, v. 236, pp. 20-29, February 1977.

11. Lan, C. E., and Roskam, J., *Airplane Aerodynamics and Performance*, p. 201, Roskam Aviation and Engineering Corporation, 1981.

64

12. Naval Surface Weapons Center, NSWC TR 84-399, *Tomahawk Land-Attack Cruise Missile Navigation, Guidance, and Flight Control,* by T. R. Pepitone and C. A. Phillips, July 1985.

13. *NATOPS Flight Manual, Navy Model A-6E TRAM/KA-6D Aircraft,* NAVAIR 01-85ADF-1, U. S. Navy, February 1986.

14. *CRC Standard Mathematical Tables,* 28th ed., p.522, CRC Press, Inc., 1988.

15. *Aerodynamics for Naval Aviators,* NAVAIR 00-80T-80, U. S. Navy, January 1965.

16. Kwak, Sehung, unpublished LISP graphics moving display program, January 1989.

17. Hurt, H. H., *Aerodynamics for Naval Aviators,* NAVAIR 00-80T-80, U.S. Government Printing Office, January 1965.

18. Hearn, Donald, and Baker, M. Pauline, *Computer Graphics,* Prentice-Hall, Inc., 1986.

19. Ross, Ron, *Planning Minimum Energy Paths in an Off-Road Environment with Anisotropic Traversal Cost and Motion Constraint,* Ph.D. Dissertation, Naval Postgraduate School, Monterey, California, June 1989.

APPENDIX A

This Appendix contains a listing of the following files:

aircraft-controls.lisp	camera.lisp
common-functions.lisp	improved-camera.lisp
intercept.lisp	kinematics.lisp
path-data.lisp	path-optimization.lisp
path-planning.lisp	setup.lisp
visibility.lisp	volume-functions.lisp
test-cases.lisp	test.lisp

Instructions for running programs:

1. Input terrain with: *(set-up 1 't27-ridges-shadow)* or *(set-up 2 't310-full-ridge)* depending on the form of the terrain file (type 1 or 2).

2. Initialize the observers with *(init-observer '(10 500 250) '0.02)*.

3. Type *(set-up-2)* to divide volumes by visibility.

4. Do search with:

    (a-star-search (init-point '(0 0 200)) (init-point '(0 1000 200)) 'nil 't))

    or

    (a-star-search-m (init-point '(0 0 200)) (init-point '(0 1000 200)) 'nil 5 't))

5. Optimize a path with: *(optimize-path '/path0002/)* or *(random-ray-optimize '(/path0002/))*.

6. To see the data on a path type *(path-data '/path0002/)*.

7. To send the data on a path to a file for the IRIS type *(path-for-iris '/path0002/)*.

```lisp
;;; -*- Mode:Common-Lisp; Base:10 -*-
;************************************************************************
;
; AIRCRAFT CONTROL              L.R. WRENN                    6 Mar 89
;
;       -----------------------------------------------------------
;
; Contains the functions nesaesary to aircraft performance.  Can be altered
; depending on the type aircraft needed and its performance spec.  The
; current aircraft is a fictional model with the following spec:
;       gross wt.                               2525 lbs.  [include full fuel]
;       desired cruise speed                    450 Kts
;       Fuel Flow (FF) Straight and level avg.  300 lbs/hour
;
;    limits on climb/dive with out gaining or loosing speed:
;       -10 deg FF - 80 lbs
;        20 deg FF - 900 lbs
;
;************************************************************************
;************************************************************************
;
; Aircraft controle routines
;
;************************************************************************
;                                        •
; takes as input the actual distance aircraft will travel
;                                                [not ground distance]
; climb angle in degrees, fuel - what you start with,
; TAS - start with.
(defun fuel-burned (distance climb-angle fuel TAS)
  (let ((FF '0)
    (original-TAS TAS)
    (climb-angle (rational climb-angle))
    (fuel-used)
    (time '0))
    (cond ((LT climb-angle -10)                 ;climb angle less than 10 deg
      (setf climb-angle '-10)
      (setf *TAS* '450)
      (setf time (* (/ distance tas) 60))
      (setf FF 80)
      (setf *fuel* (- fuel (* FF (/ time 60))))
      (setf fuel-used (* FF (/ time 60)))
      fuel-used)
     ((GT climb-angle 20)                       ;climb angle greater
                                                ; than 20 deg
      (setf TAS (get-new-TAS distance climb-angle TAS))
      (setf time (/ distance (/ (/ (+ original-TAS TAS) 2) 60)))
      (setf FF 900)
      (setf *fuel* (- fuel (* FF (/ time 60))))
      (setf fuel-used (* FF (/ time 60)))
      fuel-used)
     (t                                         ;climb angle >= -10 and <= 20
      (setf *TAS* '450)
      (setf time (* (/ distance tas) 60))
      (setf FF (+ 300
              (* 21.409090 climb-angle)
              (* .1037878 (expt climb-angle 2))
              (* .01628787 (expt climb-angle 3))))
      (setf *fuel* (- fuel (* FF (/ time 60))))
      (setf fuel-used (* FF (/ time 60)))
      fuel-used)
     )))
```

67

```lisp
; using decel rate of 3kts/degree>20/min find the new TAS
; will return new-TAS or will stop if TAS goes below 0 and return neg number
(defun get-new-TAS (dist climb-angle TAS)
  (do* ((old-time '0 new-time)
    (original-TAS TAS)
    (TAS TAS New-TAS)
    (new-time (/ dist (/ original-TAS 60))
              (/ dist (/ (/ (+ original-TAS new-TAS) 2) 60)))
        (new-TAS (- original-TAS (* (* 3 (- climb-angle 20)) new-time))
            (- original-TAS (* (* 3 (- climb-angle 20)) new-time))))
        ((or (LT (- new-time old-time) '0.05) (LT new-TAS '0)) new-tas)))



(defun t1 (climb-angle)               ; Test function used to
                                      ;    test fuel-used function
  (let ((distance '450)
      (fuel *fuel*)
      (tas *tas*)
      (time '0)
      (fuel-used '0))
    (princ distance)(terpri)
    (princ climb-angle)(terpri)
    (princ fuel)(terpri)
    (princ time)(terpri)
    (princ fuel-used)(terpri)(terpri)
    (setf fuel-used (fuel-burned distance climb-angle fuel TAS))
    (princ distance)(terpri)
    (princ climb-angle)(terpri)
    (princ fuel)(terpri)
    (princ time)(terpri)
    (princ fuel-used)(terpri)(terpri)
    ))
```

```lisp
;;; -*- Mode: LISP; Syntax: Common-lisp -*-
;********************************************************************
;;;
;;;   FLAVORS FOR 3-D DISPLAY OF VOLUMES        ;Written by Dr Sehung Kwak
;;;                                             ;for CS4452
;;;   THESIS                  D.H. Lewis                  18 May 1988
;;;
;********************************************************************

(defflavor Graphic
    (node-list
     polygon-list
     transformed-node-list
     H-matrix)
    ()
  :inittable-instance-variables
  :settable-instance-variables
  :gettable-instance-variables
  :outside-accessible-instance-variables)

(defmethod (Graphic :translate-and-euler-angle-transform)
           (x y z azimuth elevation roll)
  (let ()
       (setf H-matrix
    (homogeneous-transform azimuth elevation roll x y z))
       (setf transformed-node-list
    (mapcar #' (lambda (node-location)
               (post-multiply H-matrix node-location))
           node-list))))

(defmethod (graphic :get-node-polygon-list) ()
  (list transformed-node-list polygon-list))

(defmethod (graphic :initialize) ()
  (setf node-list (send self :make-node-list))
  (setf polygon-list (send self :make-polygon-list))
  (setf transformed-node-list node-list)
  (setf H-matrix (unit-matrix 4)))

(defmethod (graphic :get-transformed-node-list) ()
  transformed-node-list)

;********************************************************************
;;;
;;;   CAMERA FLAVOR AND METHODS TO USE GRAPHIC FLAVOR
;;;
;;;                                             ;Written by Dr Sehung Kwak
;;;                                             ;for CS4452
;;;
;;;   THESIS                  D.H. Lewis                  18 May 1988
;********************************************************************

(defflavor camera
    (H-matrix
     image-distance
     previous-point
     *camerwindow*
     scale)
    ()
  :initable-instance-variables
  :gettable-instance-variables
  :outside-accessible-instance-variables)
```

```lisp
(defmethod (camera :initialize)
       ()
  (setf H-matrix (unit-matrix 4))
  (setf image-distance *image-distance*)
  (setf scale *scale*)
  (setf *camerwindow*
    (tv:make-window 'tv:window
                  :blinker-p nil
                  :position *window-upper-left-corner*
                  :inside-width *window-width*
                  :inside-height *window-height*
                  :name "VOLUME WINDOW"
                  :save-bits t
                  :expose-p t)))

(defmethod (camera :initialize-B)      ; for advanced functions
       (window-stats)
  (setf H-matrix (unit-matrix 4))
  (setf image-distance *image-distance*)
  (setf scale *scale*)
  (setf *camerwindow*
    (tv:make-window 'tv:window
                  :blinker-p nil
                  :position (list (first window-stats)
                                    (second window-stats))
                  :inside-width (third window-stats)
                  :inside-height (fourth window-stats)
                  :name (fifth window-stats)
                  :save-bits t
                  :expose-p t)))

(defmethod (camera :move)
      (x y z azimuth elevation roll)
  (setf H-matrix (matrix-inverse
             (homogeneous-transform azimuth elevation roll x y z))))


(defmethod (camera :take-picture)
       (solid-object)
  (let* ((node-polygon-list
        (send (eval solid-object) :get-node-polygon-list))
     (node-vector (send self :convert-list-of-lists-to-vector
                       (first node-polygon-list)))
     (polygon-list (second node-polygon-list)))
;      (send *camerwindow* :refresh)        ; don't need for multiple shots
     (dolist (polygon polygon-list)
       (send self :draw-polygon polygon node-vector))))



(defmethod (camera :draw-polygon)
      (polygon node-vector)
  (let ((first-point (first polygon))
    (rest-points (cdr polygon)))
    (send self :move-pen (elt node-vector first-point))
    (dolist (point rest-points)
      (send self :draw-line (elt node-vector point)))
    (send self :draw-line (elt node-vector first-point))))
```

70

```
(defmethod (camera :move-pen)
      (point)
  (setf previous-point (send self :screen-transform point)))


(defmethod (camera :draw-line)
      (next-point)
  (let ((current-point (send self :screen-transform next-point)))
    (send self :draw-line-on-screen previous-point current-point)
    (setf previous-point current-point)))


(defmethod (camera :draw-line-on-screen)
      (from-point to-point)
  (send *camerwindow* :draw-line
          (first from-point) (second from-point)
          (first to-point) (second to-point)
          *thickness*))


(defmethod  (camera :convert-list-of-lists-to-vector)
      (list-of-lists)
  (eval (cons 'vector
          (mapcar '(lambda (component-list)
                  (cons 'list component-list))
              list-of-lists))))


(defmethod (camera :screen-transform)
      (point)
  (let* ((point-on-camerspace
        (post-multiply
          H-matrix point))
    (x (first point-on-camerspace))
    (y (second point-on-camerspace))
    (z (third point-on-camerspace)))
    (cond ((equal 0.0 z) (setf z 0.00001))
      (t))
      (list (+ (round (* scale (/ (* image-distance x) (- z))))
          (/ *window-width* 2))
      (- (/ *window-height* 2)
          (round (* scale (/ (* image-distance y) (- z)))))))))


(defmethod (camera :kill-camera-window)
      ()
  (send *camerwindow* :kill))

(defun take-picture (Camera List-of-objects)
  (send (eval Camera) :initialize)
  (send (eval Camera) :move '2000 '2000 '2000 '0 '0.5 '0.75)
  (loop for V in List-of-objects
    do (send (eval V) :initialize)
    do (send (eval V) :translate-and-euler-angle-transform '-2500
          '-2000 '-2000 '0.6 '0.6 '-0.1)
    do (send (eval Camera) :take-picture V)))
```

```
;---------------------------------------------------------------
;   advanced camera functions            D.H. Lewis
;---------------------------------------------------------------

(defvar *window-width* 700)
(defvar *window-height* 400)
(defvar *window-upper-left-corner* '(10 10))
(defvar *scale* 5)
(defvar *image-distance* 120)
(defvar *thickness* '5)                      ; line thickness, in pixels

(defvar *ideal*)
(defvar *low-left-front*)
(defvar *high-left-front*)
(defvar *low-right-front*)
(defvar *right-side*)
(defvar *left-rear-3/4*)
(defvar *top*)
(defvar *display-stats*)
(defvar *nikon-1*)
(defvar *nikon-2*)
(defvar *nikon-3*)
(defvar *nikon-4*)
(defvar *nikon-5*)
(defvar *nikon-6*)
(defvar *list-of-cameras*)
(defvar *window-stats*)

(defun make-cameras ()
  (setf *nikon-1* (make-instance 'camera))
  (setf *nikon-2* (make-instance 'camera))
  (setf *nikon-3* (make-instance 'camera))
  (setf *nikon-4* (make-instance 'camera))
  (setf *nikon-5* (make-instance 'camera))
  (setf *list-of-cameras*
   '(*nikon-1* *nikon-2* *nikon-3* *nikon-4* *nikon-5*))
  (setf
   *ideal*
   '(7500.0 3500.0 6200.0 2.0 0.0 0.9800 -500.0 -500.0 200.0 0.0 0.0 0.0))
  (setf
   *low-left-front*
   '(100.0 200.0 4000.0 0.0 0.50 1.0 1.0 1.0 -1.5 0.0 0.0 0.0))
  (setf
   *high-left-front*
   '(3725.0 -11900.0 5950.0 0.25 0.10 1.17 -500.0 -500.0 200.0 0.0 0.0 0.0))
  (setf
   *low-right-front*
   '(100.0 100.0 4000.0 0.0 0.5 1.5 1.0 1.0 1.0 0.0 0.0 0.0))
  (setf
   *right-side*
   '(00.0 -4000.0 1500.0 0.0 0.0 01.40 -500.0 -500.0 200.0 0.0 0.0 0.0))
  (setf
   *left-rear-3/4*
   '(-500.0 0.0 4000.0 0.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0))
  (setf
   *top*
   '(0.0 0.0 4000.0 0.0 0.0 0.0 -500.0 -500.0 200.0 0.0 0.0 0.0))
  'nil)
```

72

```
;----------------MAIN FOUR WINDOW DISPLAY--------------------

(defun display ()                      ;ex. (display)
    (setf *window-stats* '('nil
                    (10 20 700 400 "air-volumes" 20 140)
                    (10 440 200 200 "top-view: ground" 7 60)
                    (260 440 200 200 "same-view: ground" 20 60)
                    (510 440 200 200 "full-view: ground" 15 60)))
    (setf *display-stats* (list 'nil
                        *high-left-front*
                        *top*
                        *high-left-front*
                        *ideal*))
    (let ((air-volumes 'nil)
      (ground-volumes 'nil)
      (objects 'nil))
      (loop for V in (universe-volumes *universe*)
        do (cond ((equal 'ground (volume-composition (eval V)))
              (setf ground-volumes (adjoin V ground-volumes)))
              (t (setf air-volumes (adjoin V air-volumes)))))
        (loop for Obs in (universe-observers *universe*)
          do (setf ground-volumes (adjoin Obs ground-volumes))
          do (setf air-volumes (adjoin Obs air-volumes)))
        (setf
      objects
      (list 'nil air-volumes ground-volumes ground-volumes ground-volumes))
        (loop for N in '(1 2 3 4)
          do (take-picture-4 (nth N *list-of-cameras*)
                        (nth N *window-stats*)
                        (nth N objects)
                        (nth N *display-stats*)))))

  'nil)


;---------------DISPLAY GROUND IN (2 WINDOWS)-----------------------------

(defun display-ground ()          ;ex. (display-ground)
  (setf *window-stats* '('nil
                    (10 20 600 380 "Path-over-ground" 15 140)
                    (10 420 600 290 "Alternate-view " 20 140)
                    ('nil)
                    ('nil)))
  (setf *display-stats* (list 'nil
                        *ideal*
                        *high-left-front*
                        'nil                  ;*top*
                        'nil))                ;*right-side*))
  (let ((objects 'nil)
    (ground-volumes 'nil))
    (loop for V in (universe-volumes *universe*)
      do (cond ((equal 'ground (volume-composition (eval V)))
            (setf ground-volumes (adjoin V ground-volumes)))))
      (setf ground-volumes
        (append (universe-observers *universe*) ground-volumes))
      (setf objects (list 'nil
                    ground-volumes
                    ground-volumes
                    'nil
                    'nil))
    (loop for N in '(1 2 )
      do (take-picture-4 (nth N *list-of-cameras*)
                    (nth N *window-stats*)
```

```
                        (nth N objects)
                        (nth N *display-stats*)))))

    'nil)


;---------------DISPLAY VISIBLE AIR VOLUMES (3 WINDOWS)---------------------

(defun display-visible (observer)    ;ex. (display-visible '|observe0002|)
  (setf *window-stats* '('nil
                    (10 20 700 400 "visible-air-volumes" 20 140)
                    'nil
                    (260 440 200 200 "same-view-ground" 20 60)
                    (510 440 200 200 "full-view-ground" 15 60)))
  (setf *display-stats* (list 'nil
                    *high-left-front*
                    'nil
                    *high-left-front*
                    *ideal* ))
    (let ((visible-volumes 'nil)
    (ground-volumes 'nil)
    (objects 'nil))
    (loop for V in (universe-volumes *universe*)
      do (cond ((equal 'ground (volume-composition (eval V)))
            (setf ground-volumes (adjoin V ground-volumes))
            (setf visible-volumes (adjoin V visible-volumes)))
               ((member-p observer (volume-visibility (eval V)))
            (setf visible-volumes (adjoin V visible-volumes)))))
    (loop for Obs in (universe-observers *universe*)
        do (setf ground-volumes (adjoin Obs ground-volumes))
        do (setf visible-volumes (adjoin Obs visible-volumes)))
    (setf objects (list 'nil visible-volumes
                'nil ground-volumes ground-volumes))
    (loop for N in '(1 3 4)
      do (take-picture-4 (nth N *list-of-cameras*)
                    (nth N *window-stats*)
                    (nth N objects)
                    (nth N *display-stats*)))))
    'nil)



;---------------DISPLAY NON VISIBLE AIR VOLUMES (3 WINDOWS)----------------

(defun display-not-visible (observer)
                            ;ex. (display-not-visible '|observe0002|)
  (setf *window-stats* '('nil
                    (10 20 700 400 "non-visible-air-volumes" 20 140)
                    'nil
                    (260 440 200 200 "same-view-ground" 20 60)
                    (510 440 200 200 "full-view-ground" 15 60)))
  (setf *display-stats* (list 'nil
                    *high-left-front*
                    'nil
                    *high-left-front*
                    *ideal* ))
    (let ((invisible-volumes 'nil)
    (ground-volumes 'nil)
    (objects 'nil))
```

74

```
     (loop for V in (universe-volumes *universe*)
       do (cond ((equal 'ground (volume-composition (eval V)))
              (setf ground-volumes (adjoin V ground-volumes))
              (setf invisible-volumes (adjoin V invisible-volumes)))
             ((not (member-p observer (volume-visibility (eval V))))
              (setf invisible-volumes (adjoin V invisible-volumes)))))
     (loop for Obs in (universe-observers *universe*)
         do (setf ground-volumes (adjoin Obs ground-volumes))
         do (setf invisible-volumes (adjoin Obs invisible-volumes)))
     (setf objects (list 'nil invisible-volumes
                'nil ground-volumes ground-volumes))
     (loop for N in '(1 3 4)
       do (take-picture-4 (nth N *list-of-cameras*)
                (nth N *window-stats*)
                (nth N objects)
                (nth N *display-stats*)))))
  'nil)


;---------DISPLAY SELECTED VOLUMES AND THE GROUND (2 WINDOWS)---------------

(defun display-volumes (list-of-volumes)
          ;ex. (display-volumes '(|volume0001| |volume0012| |volume0015|))
  (setf *window-stats* '('nil
                (10 20 350 300 "desired-volumes" 17 140)
                'nil
                (510 440 200 200 "full-view-ground" 20 60)
                'nil))
  (setf *display-stats* (list 'nil
                    *high-left-front*
                    'nil
                    *high-left-front*
                    'nil))
  (let ((objects 'nil)
     (ground-volumes 'nil))
     (loop for V in (universe-volumes *universe*)
       do (cond ((equal 'ground (volume-composition (eval V)))
              (setf ground-volumes (adjoin V ground-volumes)))))
     (loop for Obs in (universe-observers *universe*)
         do (setf ground-volumes (adjoin Obs ground-volumes)))
     (setf objects (list 'nil
                list-of-volumes
                'nil
                ground-volumes
                'nil))
     (loop for N in '(1 3)
       do (take-picture-4 (nth N *list-of-cameras*)
                (nth N *window-stats*)
                (nth N objects)
                (nth N *display-stats*)))))
  'nil)
```

```
;------------------DISPLAY PATH AND GROUND (3 WINDOWS)---------------------

(defun display-path (path-name)    ;ex. (display-path '|path0002|)
  (setf *window-stats* '('nil
                   (10 20 600 380 "Path-over-ground" 15 140)
                   (10 420 600 290 "Alternate-view " 20 140)
                   (618 200 200 200 "Top-view" 7 60)
                   (618 420 200 200 "Low-side view" 7 60)))
  (setf *display-stats* (list 'nil
                         *ideal*
                         *high-left-front*
                         *top*
                         *right-side*))
  (let ((objects 'nil)
   (ground-volumes 'nil))
    (loop for V in (universe-volumes *universe*)
      do (cond ((equal 'ground (volume-composition (eval V)))
          (setf ground-volumes (adjoin V ground-volumes)))))
    (setf ground-volumes (append (universe-observers *universe*)
                         groun1-volumes))
    (setf objects (list 'nil
                   (adjoin path-name ground-volumes)
                   (adjoin path-name ground-volumes)
                   (adjoin path-name ground-volumes)
                   (adjoin path-name ground-volumes)))
    (loop for N in '(1 2 3 4)
      do (take-picture-4 (nth N *list-of-cameras*)
                 (nth N *window-stats*)
                 (nth N objects)
                 (nth N *display-stats*)))
  'nil)

(defun display-paths (list-of-paths)
                          ;ex. (display-paths '(|path0002| |path0011|))
  (setf *window-stats* '('nil
                   (10 20 600 380 "Paths-over-ground" 15 140)
                   (10 420 600 290 "Alternate-view " 20 140)
                   (618 200 200 200 "Top-view" 7 60)
                   (618 420 200 200 "Low-side view" 7 60)))
  (setf *display-stats* (list 'nil
                         *ideal*
                         *high-left-front*
                         *top*
                         *right-side*))
  (let ((objects 'nil)
   (ground-volumes 'nil))
    (loop for V in (universe-volumes *universe*)
      do (cond ((equal 'ground (volume-composition (eval V)))
          (setf ground-volumes (adjoin V ground-volumes)))))
    (setf ground-volumes (append (universe-observers *universe*)
                         ground-volumes))
    (setf objects (list 'nil
                   (append list-of-paths ground-volumes)
                   (append list-of-paths ground-volumes)
                   (append list-of-paths ground-volumes)
                   (append list-of-paths ground-volumes)))
    (loop for N in '(1 2 3 4)
      do (take-picture-4 (nth N *list-of-cameras*)
                 (nth N *window-stats*)
                 (nth N objects)
                 (nth N *display-stats*)))
  'nil)
```

```
;----------SIMPLE CAMERA ORDERS FOR A PICTURE (MANUAL CONTROL)--------------

(defun take-picture-3
        (List-of-objects x y z az roll rot ox oy oz oaz oroll orot)
   (let ((Camera '*nikon*))
   (send (eval Camera) :initialize)
   (send (eval Camera) :move x y z az roll rot )
   (loop for V in List-of-objects
     do (send (eval V) :initialize)
     do (send
           (eval V)
           :translate-and-euler-angle-transform ox oy oz oaz oroll orot)
     do (send (eval Camera) :take-picture V))))


;-----ADVANCED CAMERA ORDERS FOR A PICTURE (SEMI-AUTOMATIC CONTROL)---------

(defun take-picture-4 (Camera Window-stats List-of-objects view-stats)
   (cond ((or (null view-stats)
              (null list-of-objects))
      (return-from take-picture-4 'nil)))
   (setf *window-width* (third window-stats))
   (setf *window-height* (fourth window-stats))
   (setf *scale* (sixth window-stats))
   (setf *image-distance* (seventh window-stats))
   (send (eval Camera) :initialize-B Window-stats)
   (send (eval Camera) :move (first view-stats)        ; x
                         (second view-stats)        ; y
                    (third view-stats)         ; z
                    (fourth view-stats)        ; azimuth
                    (fifth view-stats)         ; roll
                    (sixth view-stats))        ; rotation
 (loop for V in List-of-objects
   do (send (eval V) :initialize)
   do (send
        (eval V)
        :translate-and-euler-angle-transform
        (nth 6 view-stats)
        (nth 7 view-stats)
        (nth 8 view-stats)
        (nth 9 view-stats)
        (nth 10 view-stats)
        (nth 11 view-stats))
   do (send (eval Camera) :take-picture V)
   do (let ((object-type (string-trim '"|0123456789 " V)))
        (cond ((string-equal object-type '"observer")
               (let* ((obs-point
                        (first (send
                               (eval V) :get-transformed-node-list)))
                      (screen-point (send
                                    (eval Camera)
                                    :screen-transform obs-point)))
               (send (eval (camera-*camerwindow* (eval Camera)))
                    :set-cursorpos (- (first screen-point) '30)
                    (- (second screen-point) '5))
                    (send (eval (camera-*camerwindow* (eval Camera)))
                    :display-lozenged-string '"obs")))
              ((string-equal object-type '"path")
```

```
(let* ((start-point
        (first (send (eval V)
                     :get-transformed-node-list)))
       (end-point
        (first (last
                (send
                 (eval V)
                 :get-transformed-node-list))))
       (screen-start-point
        (send (eval Camera)
              :screen-transform start-point))
       (screen-end-point
        (send (eval Camera)
              :screen-transform end-point)))
  (cond ((< '50000 (* *window-width* *window-height*))
         (send
          (eval (camera-*camerwindow* (eval Camera)))
          :set-cursorpos
          (- (first screen-start-point) '43)
          (- (second screen-start-point) '5))
         (send
          (eval (camera-*camerwindow* (eval Camera)))
          :display-lozenged-string '"start")
         (send
          (eval (camera-*camerwindow* (eval Camera)))
          :set-cursorpos
          (+ (first screen-end-point) '3)
          (- (second screen-end-point) '5))
         (send (eval (camera-*camerwindow*
                      (eval Camera)))
               :display-lozenged-string '"end")))))))))
```

78

```
;; -*- Mode:Common-Lisp; Base:10 -*-
;****************************************************************************
;
;                    C O M M O N   F U N C T I O N S
;
;            ---------------------------------------------
;
;   This file consists of all common functions used by most of the
; files of the 3-D path planning software.  Function vary from the
; very general (convenience) functions, to very detailed, special
; purpose functions (which happen to be called from two seperate
; files).  Functions are grouped by categories of Simple functions,
; Point functions, Vector functions, Line functions, Plane functions,
; Facet functions, Volume functions, property list functions,
; detailed (special purpose) functions, and finally, printing functions.
;
;   D.H.Lewis/Thesis                                    07 AUG 88
;   Modified
;   L.R.Wrenn                                           08 Apr 89
;****************************************************************************
;
;              D I R E C T O R Y    O F    F U N C T I O N S
;
;         ------------------------------------------------------------
;
; SIMPLE:   MEMBER-P              POINTS:  AVERAGE-OF-POINTS
;           EQUAL-P                        FIND-POINT
;           EQUAL-ZERO-P                   AVERAGE-POINT
;           DISTANCE
;           MERGE-JOIN-LIST
;           FIRST-NON-ZERO       VECTORS:  SOLVE-FOR-T
;           EQUAL-ERROR                    VECTOR-ADD-WITH-T
;           LT, GT, GE, LE
;
; LINES:    MAKE-LINE            PLANES:   MAKE-A-PLANE
;           LINE-CROSS-PRODUCT             MAKE-A-NORMALIZED-PLANE
;           FIND-COMMON-POINT              MAKE-VERTICAL-PLANE
;           ANGLE-BETWEEN-LINES            MAKE-Z-PLANE
;           FIND-MID-POINT                 MAKE-X-PLANE
;                                          MAKE-Y-PLANE
; FACETS:   FIND-COMMON-FACET              SUBS-POINT-INTO-EQUATION
;           MEAN-POINT-IN-A-FACET          SUBS-LINE-INTO-PLANE-EQUATION
;           MEAN-POINT-IN-A-FACET-2
;           INFO-ON-FACETS
;           INSIDE-FACET-P
;
; VOLUMES:  INTERSECT-ALL-PLANES-WITH-VOLUMES
;           INTERSECT-ALL-PLANES-WITH-VOLUMES-2
;
; PROPERTY LISTS: RESET-POINT-PROPERTY-LISTS
;
; DETAILED FUNCTIONS: MINIMUM-DISTANCE
;                     LOCATE-POINT-AIR
;                     LOCATE-POINT
;                     POINT-IN-VOLUME-P
;                     POINT-CHECK-P
;                     LINES-STRATTLE-FACETS-P
;                     SPEED-DEMON
;
```

79

```
; PRINTING FUNCTIONS: DUMP-VOLUMES
;                     DUMP-PATH
;                     PRINT-POINTS
;                     PRINT-VECTORS
;                     PRINT-LINES
;                     PRINT-FACETS
;                     PRINT-VOLUMES
;
;*****************************************************************************

(defvar *precision* '0.0025)
(defvar *precision-2* '0.25)


;------------------SIMPLE FUNCTIONS--------------


(defun member-p (item list)                     ; T or nil member
  (not (null (member item list))))

(defun equal-p (list1 list2)                    ; are two lists equal?
  (cond ((equal (length list1) (length list2))
         (apply 'and (mapcar 'equal list1 list2)))))

(defun equal-zero-p (A)                          ; is A equal to zero?
  (cond ((equal (* '1.0 A) '0.0)
    (return-from equal-zero-p 't)))
  'nil)

(defun t2 (p1 p2)
  (distance-z p1 p2))

;used to convert Z-coord from 10's of feet to NM
(setq *conversion-matrix* '((1 0 0 ) (0 1 0 ) (0 0 0.0016458195 )))


(defun distance (P1 P2)                          ; distance between two
points

  (let* ((P1 (car (matrix-multiply  (list (send (eval P1) :list-format))
                                      *conversion-matrix*)))
     (P2 (car (matrix-multiply  (list (send (eval P2) :list-format))
                                      *conversion-matrix*)))
    (difference (mapcar '- P1 P2)))
    (sqrt (apply '+ (mapcar '* difference difference)))))

(defun distance-Z (P1 P2)           ; vertical distance between two points

  (let* ((P1 (car (matrix-multiply  (list (send (eval P1) :list-format))
                                      *conversion-matrix*)))
     (P2 (car (matrix-multiply  (list (send (eval P2) :list-format))
*conversion-matrix*))))
    (- (third p2) (third P1))))

(defun climb-angle (P1 P2)
  (let ((dist-total (distance P1 P2))
    (dist-Z (distance-Z P1 P2)))
    (* (/ 180 PI) (asin (/ dist-z dist-total)))))
```

```lisp
(defun merge-join-list (List1 List2)                 ; join the two lists to make
  (let ((length1 (length list1))                     ; one long list
    (length2 (length list2))
    (templist 'nil))
    (cond ((>= length1 length2)
      (setf templist list1)
      (loop for I in list2
        do (setf templist (adjoin I templist))))
    (t (setf templist list2)
      (loop for I in list1
        do (setf templist (adjoin I templist)))))
    templist))

(defun first-non-zero (List)       ; find the first non-zero element in a simple
list
                                   ; if none found, return "-1".
  (cond ((not (equal-zero-p (first List)))
    (first List))
    ((not (equal-zero-p (second List)))
    (second List))
    ((not (equal-zero-p (third List)))
    (third List))
    (t (- 1)))))

(defun equal-error (A B)                      ; equal within an allowed level of
error
  (let ((error 'nil))
    (cond ((equal A B)                        ; simple equal
      (return-from equal-error 't))
      ((equal (* '1.0 A)                      ; floating point equal
          (* '1.0 B))
      (return-from equal-error 't))
      ((or (equal-zero-p B)                   ; divide by zero check
          (equal-zero-p A))
      (setf error '1.0))
      ((> A B)                                ; find absolute error between terms
      (setf error (abs (/ (- A B) B))))
      (t (setf error (abs (/ (- A B) A)))))
    (<= error *precision*)))                  ; check with allowed precision


(defun equal-error-2 (A B)                    ; equal within an allowed level of
error
  (let ((error 'nil))
    (cond ((equal A B)                        ; simple equal
      (return-from equal-error-2 't))
      ((equal (* '1.0 A)                      ; floating point equal
          (* '1.0 B))
      (return-from equal-error-2 't))
      ((or (equal-zero-p B)                   ; divide by zero check
          (equal-zero-p A))
      (setf error '1.0))
      ((> A B)                                ; find absolute error between terms
      (setf error (abs (/ (- A B) B))))
      (t (setf error (abs (/ (- A B) A)))))
    (<= error *precision-2*)))                ; check with allowed precision

(defun LT (A B)
  (and (not (equal-error A B))
    (< A B)))

(defun GT (A B)
```

```
(and (not (equal-error A B))
     (> A B)))
```

82

```lisp
(defun LE (A B)
  (not (GT A B)))

(defun GE (A B)
  (not (LT A B)))

;----------------MANIPULATE POINTS---------------

(defun average-of-points (list-of-points)
  (map 'list '(lambda (a b) (/ a b)) (mean-point-in-facet-2 list-of-points)
                                     (make-list 3 :initial-element
                                     (length list-of-points))))

(defun find-point (X Y Z List-of-points)  ; find all points in list which match
  (let ((result List-of-points)           ; one or more of specified values.
values
    (values (list X Y Z)))                ; of 'nil will be ignored.  returns a list.
    (loop for Pass in (List 0 1 2)
      do (cond ((not (equal 'nil (nth Pass values)))
            (let ((intermediate-result 'nil))
              (loop for P in result
                do (cond ((equal-error (nth Pass values)
                                       (nth Pass (send (eval P) :list-format)))
                      (setf intermediate-result
                            (adjoin P intermediate-result)))))
            (setf result intermediate-result)))))
    result))

(defun average-points (Pt1 Pt2) ; find the point 1/2 way between two points
  (map 'list '/ (map 'list '+ (send (eval Pt1) :list-format)
                     (send (eval Pt2) :list-format))
             (make-list 3 :initial-element '2)))




;------------MAKE OR MANIPULATE VECTORS-----------

(defun solve-for-t (Plane Line Denom)
  (/ (- (fourth Plane) (apply '+ (map 'list '* Plane
                              (send (eval(Line-segment-position-vector
                                     (eval Line))) :list-format)))) Denom))

(defun vector-add-with-t (DV PV Ti)     ; add a direction vector (*T) to a
position vector
  (map 'list '+ (send (eval PV) :list-format)
              (map 'list #'(lambda (A) (* A Ti)) (send (eval DV)
:list-format))))


;------------MAKE OR MANIPULATE LINES------------


(defun make-line (Point1 Point2)
  (init-line (init-vector '*origin* Point1)
        (init-vector Point1 Point2)))

(defun line-cross-product (L1 L2)  ; take the cross product of direction vectors
  (cross-product (send (eval (line-segment-direction-vector (eval L1)))
:list-format)
           (send (eval (line-segment-direction-vector (eval L2)))
:list-format)))
```

83

```lisp
(defun find-common-point (L1 L2)              ; returns the value of a common
point,
    (loop for m in (send (eval L1) :endpoints)   ; if one exists.
    do (loop for n in (send (eval L2) :endpoints)
            when (equal m n)
        do (return-from find-common-point m)))
    'nil)


(defun angle-between-lines (L1 L2)      ; find the smallest angle between two
lines
                                        ; return NIL for unusual problems
  (let* ((normal-vector (line-cross-product L1 L2))
     (normal-vector-length (sqrt (abs (+ (* (first normal-vector)
                                            (first normal-vector))
                          (* (second normal-vector)
                             (second normal-vector))
                          (* (third normal-vector)
                             (third normal-vector)))))))
    (cond ((equal-zero-p normal-vector-length)
       (return-from angle-between-lines 'nil))
      ((or (equal-zero-p (send (eval L1) :length))
           (equal-zero-p (send (eval L2) :length)))
       (return-from angle-between-lines 'nil)))
    (- *PI* (asin (/ normal-vector-length (* (send (eval L1) :length)
                                             (send (eval L2) :length)))))))

(defun find-mid-point (Line)
  (send (eval Line) :midpoint))



;-----------MAKE OR MANIPULATE PLANES-------------

(defun make-a-plane (point line)      ; define a plane given a point and a line
  (let* ((Obs-line (init-line (init-vector '*origin* point)
                    (init-vector point
                                 (first (send (eval line) :endpoints)))))
    (plane (make-a-normalized-plane Obs-line line)))
    (init-plane plane)))


(defun make-a-normalized-plane (L1 L2)                ; make a plane equation with
                                                      ; Ao = -1,0,1; first coef is
positive
  (let ((un-normalized (line-cross-product L1 L2))    ; normal vector to plane
    (common-point (find-common-point L1 L2))
                                        ; a point in the plane
    (Ao 'nil)                           ; constant in plane equation
    (normalized 'nil')                  ; in standard form
    (setf un-norm  .zed (map 'list 'rationalize un-normalized))
    (cond ((null common-point)
       (setf common-point (send (eval (send (eval L1) :start-point))
:list-format)))
      (t (setf common-point (send (eval common-point) :list-format)))))
    (setf Ao (apply '+ (mapcar '* common-point un-normalized)))
    (cond ((equal-zero-p Ao)
       (setf normalized
                (map 'list '/ un-normalized (make-list 3 :initial-element
                                            (first-non-zero un-normalized)))))
      (setf normalized (reverse (append (list '0) (reverse normalized)))))))
```

84

```lisp
       (t (setf normalized
           (map 'list '/ un-normalized (make-list 3 :initial-element Ao)))
           (setf normalized (reverse (append (list '1) (reverse normalized))))))))
     (cond ((GT '0.0 (first-non-zero normalized))
           (map 'list '* (make-list 4 :initial-element (- 1)) normalized))
       (t 't))
     (setf normalized (map 'list 'rationalize normalized))
     normalized))    ; return the coeffs for the plane


(defun make-vertical-plane (Line)
  (let* ((line-endpoints (send (eval Line) :endpoints))
     (Pt1 (map 'list '+ '(0 0 10)
                     (send (eval (first line-endpoints)) :list-format)))
     (L1 (make-line (init-point Pt1) (second line-endpoints)))
     (L2 (make-line (init-point Pt1) (first line-endpoints))))
     (init-plane (make-a-normalized-plane L1 L2))))


(defun make-z-plane (point)
  (init-plane (make-a-normalized-plane
                     (make-line (init-point
                                    (map 'list '+ (send (eval point ) :list-format)
                                               '(10 0 0)))
                            point)
                     (make-line (init-point
                                    (map 'list '+ (send (eval point) : list-format)
                                               '(0 10 0)))
                            point))))
(defun make-y-plane (point)
  (init-plane (make-a-normalized-plane
                     (make-line (init-point
                                    (map 'list '+ (send (eval point ) :list-format)
                                               '(0 0 10)))
                            point)
                     (make-line (init-point
                                    (map 'list '+ (send (eval point) : list-format)
                                               '(0 10 0)))
                            point))))


(defun make-x-plane (point)
  (init-plane (make-a-normalized-plane
                     (make-line (init-point
                                    (map 'list '+ (send (eval point ) :list-format)
                                               '(10 0 0)))
                            point)
                     (make-line (init-point
                                    (map 'list '+ (send (eval point) : list-format)
                                               '(0 0 10)))
                            point))))


(defun subs-point-into-equation (Plane Point)
  (apply '+ (map 'list '* (send (eval Point) :list-format) Plane)))
```

85

```
(defun subs-line-into-plane-equation (Line Plane)        ; TRUE if lines lie in
plane
   (let* ((endpoints (send (eval Line) :endpoints))
      (point-Aos (list (send (eval plane)
                              :subs-point-into-plane (first endpoints))
                        (send (eval plane)
                              :subs-point-into-plane (second endpoints)))))
     (apply 'and
       (map 'list #'equal-error
            point-Aos
            (make-list 2 :initial-element
                        (fourth (send (eval plane) :list-coeff)))))))


;----------------MANIPULATE FACETS------------------


(defun find-common-facet (V1 V2)     ; find the first facet that two volumes have
in
                                     ; common.  Use the assumption that common
facets
                                     ; will have same name first, else they will
have
                                     ; the same plane equation.
   (let ((common-facet (first (intersection (volume-facets (eval V1))
                                            (volume-facets (eval V2))))))
     (cond ((not (null common-facet))
            (return-from find-common-facet common-facet))
           ((not (null (facet-connects (eval (first (volume-facets (eval V1)))))))
            (loop for F1 in (volume-facets (eval V1))
               do (cond ((member-p V2 (second (facet-connects (eval F1))))
                         (return-from find-common-facet F1)))))
           (t (loop for F1 in (volume-facets (eval V1))
                do (loop for F2 in (volume-facets (eval V2))
                     do (cond ((send (eval F1) :test-equal F2)
                               (return-from find-common-facet F2)))))))
   'nil)


(defun mean-point-in-facet (Facet)
   (map 'list '(lambda (a b) (/ a b)) (mean-point-in-facet-2 (send (eval Facet)
:points))
                                      (make-list 3 :initial-element
                                      (length (send (eval Facet) :points)))))

(defun mean-point-in-facet-2 (Points)
   (cond ((null Points) '(0 0 0))
     (t (map 'list '+ (send (eval (first Points)) :list-format)
                     (mean-point-in-facet-2 (rest Points))))))

(defun info-on-facets (list-of-facets)    ; find all points and lines in a list
of facets
   (let ((lines 'nil)
     (points 'nil))
     (loop for F in list-of-facets
       do (let ()
         (setf lines (append (facet-edges (eval F)) lines))
         (setf points (append (send (eval F) :points) points))))
     (setf lines (remove-duplicates lines))
     (setf lines (remove 'nil lines))
     (setf points (remove-duplicates points))
     (setf points (remove 'nil points))
     (list points lines)))
```

```lisp
(defun inside-facet-p (point facet)              ; return T iff point is inside
  (let ((horizontal-plane (make-z-plane point))  ; a convex facet
    (vertical-y-plane (make-y-plane point))
    (vertical-x-plane (make-x-plane point))
    (vertical-Ao-x 'nil)
    (vertical-Ao-y 'nil)
    (left-points 'nil)
    (right-points 'nil)
    (edge-points 'nil))

                               ; intercept all edges with horizontal plane,
                               ; plane interception points in left or right
                               ; half, based upon relationship with vertical
plane

    (setf vertical-Ao-x (fourth (send (eval vertical-x-plane) :list-coeff)))
    (setf vertical-Ao-y (fourth (send (eval vertical-y-plane) :list-coeff)))
    (loop for L in (facet-edges (eval Facet))
      do (let ((I (find-intercept-point horizontal-plane L))
            (I-Ao-x 'nil)
            (I-Ao-y 'nil))
        (cond ((not (equal 'nil I))
            (setf I-Ao-y (send (eval vertical-y-plane) :subs-point-into-plane
I))
            (setf I-Ao-x (send (eval vertical-x-plane) :subs-point-into-plane
I))
            (cond ((LT vertical-Ao-x I-Ao-x)
                (setf right-points (adjoin I right-points)))
               ((GT vertical-Ao-x I-Ao-x)
                (setf left-points (adjoin I left-points)))
               (t (setf edge-points (adjoin I edge-points))))
            (cond ((LT vertical-Ao-y I-Ao-y)
                (setf right-points (adjoin I right-points)))
               ((GT vertical-Ao-y I-Ao-y)
                (setf left-points (adjoin I left-points)))
               (t (setf edge-points (adjoin I edge-points)))))))))

                        ; test for inclusion by nr of intercept points

    (cond ((or (not (evenp (length left-points)))     ; if either one odd, then
point
        (not (evenp (length right-points))))    ; is in facet
      (return-from inside-facet-p 't))
     (t (return-from inside-facet-p 'nil)))))


;-----------MAKE OR MANIPULATE VOLUMES-----------


(defun intersect-all-planes-with-volumes (list-of-planes List-of-volumes)
                        ; intersectal all planes given with all volumes given,
                        ; including resultant volumes from earlier intersections.
                        ; requires input of volumes as: ((volume) (volume) ...)
                        ; resultant volume list is the same format.
  (let ((old-list-of-error-planes 'nil)
    (result-volumes
      (intersect-all-planes-with-volumes-2 List-of-planes List-of-volumes)))
    (loop repeat '1
      do (let ()
        (terpri) (terpri)
        (princ " Re-doing error intercepts: ")
        (prin1 *list-of-error-planes*) (terpri)
```

```
            (setf old-list-of-error-planes *list-of-error-planes*)
            (setf *list-of-error-planes* 'nil)
            (setf result-volumes (intersect-all-planes-with-volumes-2
                                   old-list-of-error-planes
                                   result-volumes))))
      result-volumes))


(defun intersect-all-planes-with-volumes-2 (List-of-planes List-of-volumes)
                        ; do all the work for intersect-all-planes-with-volumes
   (let ((templist '()))
      (cond ((null list-of-planes) list-of-volumes)
         (t (loop for V in List-of-volumes
             do (let ((temp 'nil))
                  (setf temp (intersect (car V)
                                  (send (eval (car list-of-planes))
                                        :list-coeff)))
                  (cond ((equal '1 (length temp))
                         (push temp templist))
                        (t (push (list (first temp)) templist)
                           (push (list (second temp)) templist)))))
           (intersect-all-planes-with-volumes-2 (cdr list-of-planes) templist))))))


;-----------PROPERTY LIST MANIPULATIONS-----------

(defun reset-point-property-lists (Volume)
   (loop for P in (volume-points (eval Volume))
      do (setf (get P 'lines) 'nil)
      do (setf (get P 'planes) 'nil)
      do (setf (get P 'distance) 'nil)))


;-----------MANIPULATE GLOBAL COUNTERS------------

(defun speed-demon ()
   (terpri) (terpri)                              ; delete *list-of-?????* lists
to
   (princ "*******SPEED-DEMON-INVOKED*******")    ; speed processing. best if
   (terpri) (terpri)                              ; used with static universe
methods
   (setf *list-of-vectors* 'nil)                  ; if contents of old lists
still needed
   (setf *list-of-lines* 'nil)
   (setf *list-of-points* 'nil)
   (setf *list-of-planes* 'nil)
   (make-null-vector)
   (make-origin))

;---------------MORE SPECIFIC STUFF--------------

(defun minimum-distance (lines start-point)
   (let ((best-line (first lines)))
      (cond ((< '1 (length lines))
         (loop for L in (rest lines)
            do (cond ((> (get (send (eval L) :other-end start-point)
                            'distance)
                         (get (send (eval best-line) :other-end start-point)
                            'distance))
                      (setf best-line L))))))
      best-line))
```

```
;-----------------------------------------------------------
; FIND THE VOLUME(S) CONTAINING A GIVEN POINT
;-----------------------------------------------------------

(defun locate-point (point)

                ; return the one, two, or more volumes which contain the point.
                ; multiple volumes are possible if point is on facet or vertex
                ; of a volume

  (let ((list-of-possible-volumes (universe-volumes *universe*))
    (reject-volumes 'nil)
    (x-plane (make-x-plane point))
    (y-plane (make-y-plane point))
    (z-plane (make-z-plane point)))

                ; loop through planes which define point,
                ; removing volumes which do not intersect the planes.
                ; point is located in volume(s) which are left

    (loop for Pl in (list x-plane y-plane z-plane)
      do (let ()

                ; loop through (modified) list of candiate volumes

        (loop for V in list-of-possible-volumes
          do (let ((first-point-Ao (send (eval Pl) :subs-point-into-plane
                                         (first (volume-points (eval V)))))

               (Ao (fourth (send (eval Pl) :list-coeff))))

                ; see if volume strattles plane

            (cond ((not (equal-error first-point-Ao Ao))
                  (cond ((point-check-p Pl first-point-Ao Ao V)
                       (setf reject-volumes (adjoin V reject-volumes)))))))))

                ; remove rejected volumes from possible location of points

        (loop for V in reject-volumes
          do (setf list-of-possible-volumes (remove V list-of-possible-volumes)))
        (setf reject-volumes 'nil)))

                        ; select actual location of point from final list
                        ; of volumes

    (loop for V in list-of-possible-volumes     ; not so good
      do (let ((lines 'nil))
        (loop for F in (volume-facets (eval V))
          do (setf (get F 'center) (init-point (mean-point-in-facet F)))
          do (setf lines (adjoin (make-line Point (get F 'center)) lines)))
        (cond ((lines-strattle-facets-p Lines V)
              (setf list-of-possible-volumes (remove V
list-of-possible-volumes)))))))

    list-of-possible-volumes))
```

```lisp
(defun locate-point-air (point)
            ; return the one, two, or more air volumes which contain the point.
            ; multiple volumes are possible if point is on facet or vertex
            ; of a volume. Same as locate-point function, except that ground
            ; volumes are removed

  (let ((list-of-possible-volumes (universe-volumes *universe*))
        (reject-volumes 'nil)
        (x-plane (make-x-plane point))
        (y-plane (make-y-plane point))
        (z-plane (make-z-plane point)))

                    ; loop through planes which define point,
                    ; removing volumes which do not intersect the planes.
                    ; point is located in volume(s) which are left

    (loop for Pl in (list x-plane y-plane z-plane)
      do (let ()

                    ; loop through (modified) list of candiate volumes

        (loop for V in list-of-possible-volumes
          do (let ((first-point-Ao (send (eval Pl) :subs-point-into-plane
                                            (first (volume-points (eval V)))))

                   (Ao (fourth (send (eval Pl) :list-coeff))))

                    ; see if volume strattles plane

              (cond ((not (equal-error first-point-Ao Ao))
                     (cond ((point-check-p Pl first-point-Ao Ao V)
                            (setf reject-volumes (adjoin V reject-volumes)))))))))

                    ; remove rejected volumes from possible location of points

        (loop for V in reject-volumes
          do (setf list-of-possible-volumes (remove V list-of-possible-volumes)))
        (setf reject-volumes 'nil)))

                    ; select actual location of point from final list
                    ; of volumes

    (loop for V in list-of-possible-volumes     ; not so good
      do (let ((lines 'nil))
        (loop for F in (volume-facets (eval V))
          do (setf (get F 'center) (init-point (mean-point-in-facet F)))
          do (setf lines (adjoin (make-line Point (get F 'center)) lines)))
        (cond ((lines-strattle-facets-p Lines V)
               (setf list-of-possible-volumes (remove V
list-of-possible-volumes))))))


                    ; remove ground volumes from list

    (loop for V in list-of-possible-volumes
      do (cond ((equal 'ground (volume-composition (eval V)))
               (setf list-of-possible-volumes (remove V list-of-possible-volumes)))))

    list-of-possible-volumes))
```

```lisp
(defun point-in-volume-p (point volume) ; return T iff the point is inside the
volume
                                        ; return NIL otherwise
                                        ; code is modified version of
locate-point-air
  (let ((list-of-possible-volumes (list volume))
    (reject-volumes 'nil)
    (x-plane (make-x-plane point))
    (y-plane (make-y-plane point))
    (z-plane (make-z-plane point)))

                  ; see if point is a vertex, or in a facet of the volume

    (cond ((member-p point (volume-points (eval volume)))
      (return-from point-in-volume-p 't)))
    (loop for F in (volume-facets (eval volume))
      do (cond ((inside-facet-p point F)
          (return-from point-in-volume-p 't))))

                  ; loop through planes which define point,
                  ; removing volumes which do not intersect the planes.
                  ; point is located in volume(s) which are left

    (loop for Pl in (list x-plane y-plane z-plane)
      do (let ()

              ; loop through (modified) list of candiate volumes

        (loop for V in list-of-possible-volumes
          do (let ((first-point-Ao (send (eval Pl) :subs-point-into-plane
                                          (first (volume-points (eval V))))))

            (Ao (fourth (send (eval Pl) :list-coeff))))

                  ; see if volume strattles plane

            (cond ((not (equal-error first-point-Ao Ao))
                (cond ((point-check-p Pl first-point-Ao Ao V)
                    (setf reject-volumes (adjoin V reject-volumes)))))))))

              ; remove rejected volumes from possible location of points

        (loop for V in reject-volumes
          do (setf list-of-possible-volumes (remove V list-of-possible-volumes)))
        (setf reject-volumes 'nil)))

      (cond ((null list-of-possible-volumes)                    ; exit condition
        (return-from point-in-volume-p 'nil)))

    't))

(defun point-check-p (Plane Basis-point-Ao Ao Volume)
  (loop for P in (rest (volume-points (eval Volume)))
    do (let ((next-point-Ao (send (eval Plane) :subs-point-into-plane P)))
    (cond ((equal next-point-Ao Ao)
          (return-from point-check-p 'nil))
          ((or (and (GT Ao Next-point-Ao)
              (LT Ao basis-point-Ao))
            (and (LT Ao Next-point-Ao)
              (GT Ao basis-point-Ao)))
          (return-from point-check-p 'nil)))))
  't)
```

91

```
(defun lines-strattle-facets-p (Lines Volume)
  (loop for L in Lines
    do (loop for F in (volume-facets (eval Volume))
    do (cond ((send (eval L) :strattle-plane-p F)
             (return-from lines-strattle-facets-p 't)))))
  'nil)


;-----------------------------------------------------------
;  PRINT GOOD-TO-KNOW INFO CONCERNING THE STATE
;  OF THE *UNIVERSE* INTO A DISK FILE
;-----------------------------------------------------------

(defun dump-volumes (list-of-volumes)
  (setq *output-stream* (open "exp3:lewis;run2" :direction :output))
  (print "sending data to file (run2)...")
  (loop for V in List-of-volumes
     do (let ()
     (terpri *output-stream*) (terpri *output-stream*) (terpri *output-stream*)
     (print-volumes (list V))
     (terpri *output-stream*)
     (print-points (volume-points (eval V)))
     (terpri *output-stream*)
     (print-lines (volume-edges (eval V)))
     (terpri *output-stream*)
     (print-facets (volume-facets (eval V)))))
  (terpri *output-stream*) (terpri *output-stream*) (terpri *output-stream*)
  (close *output-stream*)
  (print "Done.") 'nil)


(defun dump-path (path-name)
  (setq *output-stream* (open "exp3:lewis;path-dump" :direction :output))
  (print "sending path data to file (path-dump)...")
  (terpri *output-stream*) (terpri *output-stream*) (terpri *output-stream*)
  (print-path path-name)
  (terpri *output-stream*)
  (print-points (path-points (eval path-name)))
  (terpri *output-stream*)
  (print-lines (path-lines (eval path-name)))
  (terpri *output-stream*)
  (print-facets (path-facets (eval path-name)))
  (terpri *output-stream*) (terpri *output-stream*) (terpri *output-stream*)
  (close *output-stream*)
  (print "Done.")
'nil)
```

92

```
(defun print-points (List)
  (cond ((null List))
    (t (terpri *output-stream*)
       (prin1 "name: " *output-stream*)
          (prin1 (car List) *output-stream*)
       (send (eval (car List)) :print)
       (print-points (cdr List)))))


(defun print-vectors (List)
  (cond ((null List))
    (t (terpri *output-stream*)
       (prin1 "name: " *output-stream*)
          (prin1 (car List) *output-stream*)
       (send (eval (car List)) :print)
       (print-vectors (cdr List)))))


(defun print-lines (List)
  (cond ((null List))
    (t (terpri *output-stream*)
       (prin1 "name:" *output-stream*)
          (prin1 (car List) *output-stream*)
       (send (eval (car List)) :print)
       (print-lines (cdr List)))))


(defun print-facets (List)
  (cond ((null List))
    (t (terpri *output-stream*)
       (prin1 "name:" *output-stream*)
          (prin1 (car List) *output-stream*)
       (send (eval (car List)) :print)
       (print-facets (cdr List)))))


(defun print-volumes (List)
  (cond ((null List))
    (t (terpri *output-stream*)
       (prin1  "name:" *output-stream*)
          (prin1 (car List) *output-stream*)
       (send (eval (car List)) :print)
       (print-volumes (cdr List)))))
```

```
(defun print-path (name)
  (terpri *output-stream*)
  (princ "name: " *output-stream*) (prinl name *output-stream*)
  (princ "start-point: " *output-stream*)
  (prinl (path-start-point (eval name)) *output-stream*)
  (terpri *output-stream*)
  (princ "end-point: "*output-stream*)
  (prinl (path-end-point (eval name)) *output-stream*)
  (terpri *output-stream*)
  (princ "volumes: "*output-stream*)
  (prinl (path-volumes (eval name)) *output-stream*)
  (terpri *output-stream*)
  (princ "facets: "*output-stream*)
  (prinl (path-facets (eval name)) *output-stream*)
  (terpri *output-stream*)
  (princ "lines: "*output-stream*)
  (prinl (path-lines (eval name)) *output-stream*)
  (terpri *output-stream*)
  (princ "points: "*output-stream*)
  (prinl (path-points (eval name)) *output-stream*)
  (terpri *output-stream*)
  (princ "length: "*output-stream*)
  (prinl (path-length (eval name)) *output-stream*)
  (terpri *output-stream*)
  (princ "total K values: "*output-stream*)
  (prinl (path-total-K (eval name)) *output-stream*)
  (terpri *output-stream*)
  (princ "maximum detection probability: "*output-stream*)
  (prinl (path-max-detection-probability (eval name)) *output-stream*)
  (terpri *output-stream*)
  (princ "average detection probability: "*output-stream*)
  (prinl (path-ave-detection-probability (eval name)) *output-stream*)
  (terpri *output-stream*))
```

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: USER -*-
;
;*********************************************************************************
;;;
;;;    MOVIE-CAMERA FLAVORS AND METHODS              ;Written by Dr Sehung Kwak
;;;                                                  ;Mod for speed by Mark Kindl
;;;
;;;    THESIS              L.R. WRENN                        12 Mar 1989
;;;
;;;    Additions and Mods for Thesis and CS-4313
;;;
;*********************************************************************************
; Improved-Movie-Camera
;
;                          FLAVORS AND METHODS
;
;
;          --------------------------------------------------------
;
; FLAVOR:    ..........................Movie-camera
;
; METHODS:  :initialize       ;set-up for movie-camera
;           :initialize-B     ;set-up for movie-camera used by advanced functions
;           :move             ;sets H-matrix for movie-camera
;           :show             ;displays an object using movie-camera
;                             NOTE:  clear-scene removed to show multi-objects
;           :clear-scene      ;refreshes (clears) the non-visible
;                                                 window of movie-camera
;           :make-visible     ;does bitblt of back-window to front-window
;           :draw-line        ;draws line to back-window
;           :kill             ;removes both windows
;           :screen-transform ;transforms real-world
;                                         list-of-points to screen-coords
;           :display-label    ;allows for labeling of objects on the screen
;
;
;*********************************************************************************
*
;
;
;                 D I R E C T O R Y   O F   F U N C T I O N S
;
;
;          ----------------------------------------------------------------------
;
;   make-movie-cameras
;   reset-window-stats
;   movie-ground
;   movie-ground-path
;   show-path-4              ;does not reset windows only adds path and ground
;   show-movie-4
;
;*********************************************************************************

(defflavor movie-camera
   (H-matrix image-distance previous-point scale
           *movie-display-window* *movie-window* *movie-window-array*)
   ()
  :initable-instance-variables)
```

```
(defmethod (movie-camera :initialize)
       ()
  (setf H-matrix '((1 0 0 0) (0 1 0 0) (0 0 1 0) (0 0 0 1)) )
  (setf image-distance *image-distance* )
  (setf scale *scale*)
  (setf *movie-display-window*
    (tv:make-window 'tv:window
                  :blinker-p nil
                  :position *movie-window-position*
                  :inside-width *movie-window-inside-width*
                  :inside-height *movie-window-inside-height*
                  :name "movie-display-window"
                  :save-bits t
                  :expose-p t))
  (setf *movie-window*
    (tv:make-window 'tv:window
                  :blinker-p nil
                  :position *movie-window-position*
                  :inside-width *movie-window-inside-width*
                  :inside-height *movie-window-inside-height*
                  :name "movie-window"
                  :save-bits t
                  :expose-p nil))
  (setf *movie-window-array*
    (send *movie-window* :bit-array)))


(defmethod (movie-camera :initialize-B)   ;for advanced functions
       (window-stats)
  (setf H-matrix '((1 0 0 0) (0 1 0 0) (0 0 1 0) (0 0 0 1)) )
  (setf image-distance *image-distance*)
  (setf scale *scale*)
  (setf *movie-display-window*
    (tv:make-window 'tv:window
                  :blinker-p nil
                  :position (list (first window-stats)
                                  (second window-stats))
                  :inside-width (third window-stats)
                  :inside-height (fourth window-stats)
                  :name (fifth window-stats)
                  :save-bits t
                  :expose-p t))
  (setf *movie-window*
    (tv:make-window 'tv:window
                  :blinker-p nil
                  :position (list (first window-stats)
                                  (second window-stats))
                  :inside-width (third window-stats)
                  :inside-height (fourth window-stats)
                  :name (fifth window-stats)
                  :save-bits t
                  :expose-p nil))
  (setf *movie-window-array*
    (send *movie-window* :bit-array)))


(defmethod (movie-camera :move)
       (x y z azimuth elevation roll)
  (setf H-matrix (matrix-inverse
     (homogeneous-transform azimuth elevation roll x y z))))
```

96

```
(defmethod (movie-camera :show)
      (solid-object)
  (let* ((node-polygon-list (send solid-object :get-node-polygon-list))
         (screen-vector (send self :screen-transform (first node-polygon-list)))
      (polygon-list (second node-polygon-list)) )
;   (send self :clear-scene)   not needed for multi object picture
      (dolist (polygon polygon-list)
        (send self :draw-polygon polygon screen-vector))
      (send self :make-visible)))


(defmethod (movie-camera :clear-scene)
      ()
  (tv:sheet-force-access (*movie-window*)
    (send *movie-window* :refresh)))


(defmethod (movie-camera :draw-polygon)
      (polygon screen-vector)
  (let ((first-point (first polygon))
    (rest-points (cdr polygon)))
     (setf previous-point (elt screen-vector first-point))
     (dolist (point rest-points)
       (send self :draw-line (elt screen-vector point)))
     (if (> (length polygon) 2)
       (send self :draw-line (elt screen-vector first-point)) )))


(defmethod (movie-camera :make-visible)
      ()
  (send *movie-display-window* :bitblt
    tv:alu-seta
    *movie-window-inside-width*            '
    *movie-window-inside-height*
    *movie-window-array*
    2 2 0 0))


(defmethod (movie-camera :draw-line)
      (next-point)
  (let ((current-point next-point))
       (tv:sheet-force-access (*movie-window*)
         (send *movie-window* :draw-line
            (first previous-point) (second  previous-point)
            (first current-point)  (second current-point)  )
       (setf previous-point current-point)) ))


(defmethod (movie-camera :kill)
      ()
  (send *movie-display-window* :kill)
  (send *movie-window* :kill))
```

97

```
(defmethod (movie-camera :screen-transform)
           (node-vector)
  (do* ((point-list node-vector (cdr point-list))
    (camera-point nil)
    (point nil)
    (z nil)
        (screen-vector nil) )
       ((null point-list) screen-vector)
       (setf point (car point-list))
       (setf camera-point (post-multiply H-matrix point))
       (setf z (* -1 (third camera-point)))
       (cond ((equal 0.0 z) (setf z 0.00001))
         (t))
    (setf screen-vector (append screen-vector (list (list
          (+ (round (* scale (/ (* image-distance
                                  (first  camera-point)) z)))
            (/ *movie-window-inside-width* 2))
          (- (/ *movie-window-inside-height* 2)
            (round (* scale (/ (* image-distance
                                  (second camera-point)) z)))))))))))))

(defmethod (movie-camera :display-label)    ;allows for the addition
                                    ; of labels to display
      (V)
  (let ((object-type (string-trim '"|0123456789 " V)))
    (cond ((string-equal object-type '"observer")
      (let* ((obs-point (first (send (eval V) :get-transformed-node-list)))
             (screen-point (car(send self
                                :screen-transform (list obs-point)))))
        (tv:sheet-force-access (*movie-window*)
          (send *movie-window* :set-cursorpos
              (- (first screen-point) '30)
              (- (second screen-point) '5))
          (send *movie-window* :display-lozenged-string '"obs")
          )))
     ((string-equal object-type '"path")
      (let* ((start-point (first (send (eval V)
                                  :get-transformed-node-list)))
             (end-point (first (last (send (eval V)
                                      :get-transformed-node-list))))
             (screen-start-point (car(send self :screen-transform
                                       (list start-point))))
             (screen-end-point   (car(send self :screen-transform
                                       (list end-point  )))))
        (tv:sheet-force-access (*movie-window*)
          (cond ((< '10000 (* *movie-window-inside-width*
                             *movie-window-inside-height*))
              (send *movie-window* :set-cursorpos
                  (- (first screen-start-point) '43)
                  (- (second screen-start-point) '5))
              (send *movie-window* :display-lozenged-string '"start")
              (send *movie-window* :set-cursorpos
                  (+ (first screen-end-point) '3)
                  (- (second screen-end-point) '5))
              (send *movie-window*
                  :display-lozenged-string '"end"))))))))
  (send self :make-visible))
```

98

```
;-------------------------------------------------------------------------
;  advanced movie-camera functions                          L. R. WRENN
;-------------------------------------------------------------------------
;*************************************************************************
;  All items commented out here are also defined in camera
;*************************************************************************

(defvar *movie-window-inside-width* 300)
(defvar *movie-window-inside-height* 300)
(defvar *movie-window-position* '(10 10))
;(defvar *scale* 10)
;(defvar *image-distance* 20)
;(defvar *thickness* '5)                        ; line thickness, in pixels

;(defvar *ideal*)
;(defvar *low-left-front*)
;(defvar *high-left-front*)
;(defvar *low-right-front*)
;(defvar *right-side*)
;(defvar *left-rear-3/4*)
;(defvar *top*)
;(defvar *display-stats*)
(defvar *rca-1*)
(defvar *rca-2*)
(defvar *rca-3*)
(defvar *rca-4*)
(defvar *rca-5*)
(defvar *rca-6*)
(defvar *list-of-vcrs*)
;(defvar *window-stats*)

(defun make-movie-cameras ()
  (setf *rca-1* (make-instance 'movie-camera))
  (setf *rca-2* (make-instance 'movie-camera))
  (setf *rca-3* (make-instance 'movie-camera))
  (setf *rca-4* (make-instance 'movie-camera))
  (setf *rca-5* (make-instance 'movie-camera))
  (setf *list-of-vcrs* '(*rca-1* *rca-2*  *rca-3* *rca-4* *rca-5*))
;   (setf *ideal* '(7500.0 3500.0 6200.0 2.0 0.0 0.9800 -500.0 -500.0 200.0 0.0
0.0 0.0))
;   (setf *low-left-front* '(100.0 200.0 4000.0 0.0 0.50 1.0 1.0 1.0 -1.5 0.0 0.0
0.0))
;   (setf *high-left-front* '(3500.0 -11900.0 5700.0 0.26 0.10 1.17
;                                     -500.0 -500.0 200.0 0.0 0.0 0.0))
;   (setf *low-right-front* '(100.0 100.0 4000.0 0.0 0.5 1.5 1.0 1.0 1.0 0.0 0.0
0.0))
;   (setf *right-side* '(00.0 -4000.0 1500.0 0.0 0.0 01.40
;                             -500.0 -500.0 200.0 0.0 0.0 0.0))
;   (setf *left-rear-3/4* '(-500.0 0.0 4000.0 0.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0
0.0))
;   (setf *top* '(0.0 0.0 4000.0 0.0 0.0 0.0 -500.0 -500.0 200.0 0.0 0.0 0.0))
  'nil)



(defun reset-window-stats (window-stats) ;used to move from one window to
another
  (setf *movie-window-inside-width* (third window-stats))
  (setf *movie-window-inside-height* (fourth window-stats))
  (setf *scale* (sixth window-stats))
  (setf *image-distance* (seventh window-stats)))
```

99

```
;---------------DISPLAY GROUND IN (4 WINDOWS)---------------------------

(defun movie-ground ()  ;initializes and displays the ground in 4 views for
paths
   (setf *window-stats* '('nil
                  (10   10 400 380 "Path-over-ground" 15 140)
                  (10 410 400 290 "Alternate-view "  20 140)
                  (420 10 300 380 "Top-view" 5 140)
                  (420 410 300 290 "Top-view No Reset of Paths" 5 140)))
   (setf *display-stats* (list 'nil
                  *ideal*
                  *high-left-front*
                  *top*
                  *top*))
   (let ((objects 'nil)
    (ground-volumes 'nil))
     (loop for V in (universe-volumes *universe*)
       do (cond ((equal 'ground (volume-composition (eval V)))
             (setf ground-volumes (adjoin V ground-volumes)))))
     (setf ground-volumes (append (universe-observers *universe*)
ground-volumes))
     (setf objects (list 'nil
                  ground-volumes
                  ground-volumes
                  ground-volumes
                  ground-volumes ))

     (loop for N in '(1 2 3 4 )
       do (show-movie-4 (nth N *list-of-vcrs*)
                  (nth N *window-stats*)
                  (nth N objects)
                  (nth N *display-stats*)))))

   'nil)


(defun movie-ground-path (path-name)         ; displays the ground and
                                        ;     path just like camera
   (setf *window-stats* '('nil
                  (10   10 400 380 "Path-over-ground" 15 140)
                  (10 410 400 290 "Alternate-view "  20 140)
                  (420 10 300 380 "Top-view" 5 140)
                  (420 410 300 290 "Low-side view" 5 140)))
   (setf *display-stats* (list 'nil
                  *ideal*
                  *high-left-front*
                  *top*
                  *right-side*))
   (let ((objects 'nil)
    (ground-volumes 'nil))
     (loop for V in (universe-volumes *universe*)
       do (cond ((equal 'ground (volume-composition (eval V)))
             (setf ground-volumes (adjoin V ground-volumes)))))
     (setf ground-volumes (append (universe-observers *universe*)
ground-volumes))
     (setf objects (list 'nil
                  (append  ground-volumes (list path-name))
                  (append  ground-volumes (list path-name))
                  (append  ground-volumes (list path-name))
                  (append  ground-volumes (list path-name))))
```

```
      (loop for N in '(1 2 3 4 )
        do (show-movie-4 (nth N *list-of-vcrs*)
                         (nth N *window-stats*)
                         (nth N objects)
                         (nth N *display-stats*)))))

  'nil)


               ; display-movie-path used by search routines to display the
               ; the search as it is on going.  Helpful when altering
parameters
               ; and observing their effect.  Note: It is a center of volume to
               ; center of volume display.
(defun display-movie-path (agenda start-point ground-volumes)
  (let ((current-best-path)
    (temp-best-path 'nil)
    (temp-path-volumes 'nil)
    (temp-goal-volume 'nil))

          (princ "-----New Agenda Sent to Movie-camera---")(terpri)
          (setf current-best-path (car agenda))
          (setf temp-goal-volume (car (agenda-item-path
                                        (eval current-best-path))))

          (setf temp-path-volumes (reverse (agenda-item-path
                                              (eval current-best-path))))
          (setf temp-best-path (init-new-path start-point
                                      (volume-arithmetic-center
                                       (eval temp-goal-volume))
                                      temp-path-volumes
                                      'nil
                                      'nil
                                      'nil
                                      'nil
                                      'nil))
          (make-center-to-center-path temp-best-path)
          (calc-path-and-stats temp-best-path)
          (send (eval temp-best-path) :initialize)
          (loop for N in '(1 2 3 4)
                do (cond ((not (equal N 4))
                       (send (eval (nth N *list-of-VCRs*))
                             :clear-scene)))
                do (show-path-4 (nth N *list-of-VCRs*)
                             (nth N *window-stats*)
                             temp-best-path
                             (first ground-volumes)
                             (nth N *display-stats*))
                do (cond ((not (equal N 4))
                       (send (eval (nth N *list-of-VCRs*))
                             :display-label temp-best-path))))))))
```

101

```
;-----ADVANCED MOVIE-CAMERA ORDERS FOR A PICTURE (SEMI-AUTOMATIC
CONTROL)--------

(defun show-path-4 (VCR Window-stats path ground view-stats)
  (reset-window-stats Window-stats)
  (send (eval path) :translate-and-euler-angle-transform (nth 6 view-stats)
                                                          (nth 7 view-stats)
                                                          (nth 8 view-stats)
                                                          (nth 9 view-stats)
                                                          (nth 10 view-stats)
                                                          (nth 11 view-stats))

  (send (eval VCR) :show (eval ground))
  (send (eval VCR) :show (eval path)))
```

```
;;; -*- Mode:Lisp; Syntax: Common-lisp -*-
;**********************************************************************
;
;     FUNCTIONS TO INTERCEPT A VOLUME WITH A PLANE          D.H.LEWIS    27May88
;
;--------------------------------------------------------------------;
;These functions intercept planes with volumes and lines with planes.  Multiple
; tests are performed to ensure proper construction of new volumes.  Facets are
; rebuilt each time.
;
; Main functions:  INTERSECT (VOLUME PLANE)
;                  FIND-INTERCEPT-POINT (PLANE LINE)
;
; Other functions: GET-INTERCEPT-POINT (PLANE LINE T-INTERCEPT)
;                  PUT-LINE-IN-CORRECT-HALF (LINE PLANE)
;                  PUSH-ENDPOINTS (LINE VOLUME)
;                  MAKE-NEW-DIVIDING-LINES (VOLUME OLDPOINTS NEW-POINTS)
;                  NEW-VALID-LINE (POINT1 POINT2 VOLUME)
;                  IN-FACET-P (POINT1 POINT2 FACET)
;                  OUTSIDE-VOLUME (LINE VOLUME)
;                  DENOM-IN-INTERCEPT (PLANE LINE)
;                  GET-INTERCEPT-POINT-2 (LINE T-INTERCEPT)
;
;**********************************************************************

(defvar *lines-in-intercept-plane* 'nil)
(defvar *large-integer* '1e4)
(defvar *list-of-error-planes* 'nil)     ; used to correct errors in
interceptions

(defun intersect (Volume Plane)
  (let ((old-precision *precision*)
    (bad-euler-flag 't)
    (new-volume1 'nil)
    (new-volume2 'nil)
    (facet-planes 'nil)
    (intercept-plane 'nil))
    (terpri) (princ "intersecting: ") (prin1 (list Volume Plane))
    (princ " --- Result: ")
    (setf *lines-in-intercept-plane* 'nil)
    (cond ((bad-intersect-preconditions-p Volume Plane); check for degenerate
conditions
       (return-from intersect (list volume))))
    (setf intercept-plane (init-plane Plane))
    (loop for F in (volume-facets (eval Volume))            ; get all planes used
      do (setf facet-planes (adjoin (init-plane (send (eval F) :list-coeff))
                     facet-planes)))
    (setf facet-planes (adjoin intercept-plane facet-planes))
    (setf facet-planes (remove-duplicates facet-planes))
    (loop until (or (not bad-euler-flag) (> *precision* (* '5 old-precision)))
      do (let ()
                         ; clear standard volumes before use (or reuse)
                         ; and set common values

        (send *above* :clear)
            (setf (volume-visibility *above*) (volume-visibility (eval Volume)))

            (setf (volume-composition *above*) (volume-composition (eval
Volume)))
            (send *below* :clear)
            (setf (volume-visibility *below*) (volume-visibility (eval Volume)))
```

103

```lisp
                (setf (volume-composition *below*) (volume-composition (eval
Volume)))

                                  ; conduct intercept

            (let ((List-of-new-points 'nil)
              (list-of-old-points 'nil))
              (loop for P in (volume-points (eval Volume))
                do (setf (get P 'lines) 'nil))
                                    ;intersect each line of volume
            (loop for Line in (Volume-Edges (eval Volume))
              do (let ((new-point (find-intercept-point intercept-plane
Line)))
                (cond ((equal new-point 'nil)
                    (cond ((not (subs-line-into-plane-equation Line
                                                intercept-plane))
                        (put-line-in-correct-half
                            Line
                            (first (send (eval Line) :endpoints))
                            intercept-plane))))
                  ((member-p new-point (Volume-points (eval Volume)))
                   (pushnew new-point list-of-old-points)
                   (put-line-in-correct-half Line new-point intercept-plane))
                  (t (pushnew new-point List-of-new-points)
                     (place-intercept-point Plane Line New-point)))))

            (make-new-dividing-lines Volume List-of-new-points
list-of-old-points))
          (cond ((not (simple-volume-test-p))          ; check degenerate cases
             (setf *precision* old-precision)
             (return-from intersect (list volume))))

                                  ; build new facets in best way possible

          (cond ((not *not-convex-volumes*)            ; do convex facets
             (make-facets facet-planes *above*)    ; quick facet builder
             (make-facets facet-planes *below*)
             (cond ((not (check-eulers-relation-p))
                (setf (volume-facets *above*) 'nil)
                (setf (volume-facets *below*) 'nil)
                (make-all-facets *above*)        ; slow facet builder
                (make-all-facets *below*))))
            (t (make-all-facets *above*)              ; do non-convex facets
               (make-all-facets *below*)))

          (cond ((null (intersection (volume-facets *above*)
                            (volume-facets *below*)))
             (force-facet plane)))

          (cond ((not (check-eulers-relation-p))
              (setf *precision* (* *precision* '2.0)))
             (t (setf bad-euler-flag 'nil)))))

    (cond ((not bad-euler-flag)
      (setf new-volume1 (make-volume-name))      ; give legitimate names to new
          (setf new-volume2 (make-volume-name))    ; volumes
          (send *above* :make-equal new-volume1)
          (send *below* :make-equal new-volume2)
          (push  new-volume1 *list-of-volumes*)
          (push  new-volume2 *list-of-volumes*)
          (setf *precision* old-precision)
          (prin1 (intersection (volume-facets (eval new-volume1))
```

104

```
                              (volume-facets (eval new-volume2))))
                  (return-from intersect (list new-volume1 new-volume2)))  ; return
new volumes
          (t (setf *precision* old-precision)
             (setf *list-of-error-planes*
                  (adjoin intercept-plane *list-of-error-planes*))
             (return-from intersect (list Volume))))))

(defun subs-point-into-plane (Pt Plane)
  (send (eval Plane) :subs-point-into-plane Pt))

(defun bad-intersect-preconditions-p (Volume Plane)  ; test for null plane (0 0
0 0)
                                                   ; and facet intercept if
convex
   (cond ((equal *zero-vector* (map 'list '* plane *one-vector*))
      (princ "nil (early 1)")
      (return-from bad-intersect-preconditions-p 't))
;    ((not *not-convex-volumes*)
;     (loop for F in (volume-facets (eval Volume))
;        do (cond ((send (eval F) :test-equal (init-plane Plane))
;                 (princ "nil (early 2)")
;                 (return-from bad-intersect-preconditions-p 't)))))
    )
  'nil)

(defun find-intercept-point (plane line) ; find intercept point of plane and
line
                                      ; segment, if it exists.  return NIL
                                      ; if not exist
  (let ((denom (rationalize (denom-in-intercept plane line)))
    (t-intercept 'nil)
    (I-point 'nil))
    (cond ((not (equal-zero-p denom))
       (setf t-intercept (rationalize (solve-for-t
                                      (send (eval plane) :list-coeff)
                                      line
                                      denom)))
       (setf I-point (get-intercept-point-2 line t-intercept))))
    I-point))

(defun denom-in-intercept (plane line) ; find the denominator in intercept
equation
   (apply '+ (map 'list '* (send (eval plane) :list-coeff)
                   (map 'list 'rationalize
                        (send (eval (line-segment-direction-vector
                                     (eval line))) :list-format)))))

(defun get-intercept-point-2 (line t-intercept)
                                          ; return the name of a valid intercept
point
  (let ((I 'nil)
    (I-list 'nil))
    (cond ((not (or (GT t-intercept (line-segment-t-max (eval line)))
                (LT t-intercept '0.0)))
       (setf I-list (vector-add-with-t
                        (line-segment-direction-vector (eval line))
                        (line-segment-position-vector (eval line))
                        t-intercept))
       (setf I (init-point I-list))))
    I))
```

105

```lisp
(defun place-intercept-point (Plane Line I) ; divide old line at I, build new
lines
   (let ((L1 'nil)                          ; and put each in right resultant
volume
     (L2 'nil))
       (setf (get I 'lines) Line)
       (pushnew I (volume-points *above*))
       (pushnew I (volume-points *below*))
       (setf L1 (make-line I (first (send (eval Line) :endpoints))))
       (setf L2 (make-line I (second (send (eval Line) :endpoints))))
       (setf (line-segment-characteristics (eval L1))      ; ridge is still a
ridge
                    (line-segment-characteristics (eval Line)))
       (setf (line-segment-characteristics (eval L2))
                    (line-segment-characteristics (eval Line)))
       (cond ((LT (fourth Plane)       ; which volume to put new line L1?
                (subs-point-into-equation Plane
                                          (car (send (eval Line) :endpoints
                                               ))))
          (pushnew L1 (volume-edges *above*))
          (push-endpoints L1 '*above*))
         ((GT (fourth Plane)
                (subs-point-into-equation Plane
                                          (car (send (eval Line) :endpoints
                                               ))))
          (pushnew L1 (volume-edges *below*))
          (push-endpoints L1 '*below*))
         (t ))
       (cond ((LT (fourth Plane)       ; Which volume to put new line L2?
                (subs-point-into-equation Plane
                                          (cadr (send (eval Line) :endpoints
                                               ))))
          (pushnew L2 (volume-edges *above*))
          (push-endpoints L2 '*above*))
         ((GT (fourth Plane)
                (subs-point-into-equation Plane
                                          (cadr (send (eval Line) :endpoints
                                               ))))
          (pushnew L2 (volume-edges *below*))
          (push-endpoints L2 '*below*)))))

(defun put-line-in-correct-half (Line Point Plane) ; put a preexisting volume
line
                                                  ; into the correct resultant
volume
   (let ((Plane-Ao (fourth (send (eval Plane) :list-coeff)))
     (other-point (send (eval Line) :other-end Point)))
     (cond ((GT (send (eval Plane) :subs-point-into-plane other-point)
                Plane-Ao)
               (pushnew Line (volume-edges *above*))
               (push-endpoints Line '*above*))
       (t (pushnew Line (volume-edges *below*))
          (push-endpoints Line '*below*)))))


(defun push-endpoints (Line Volume)
   (pushnew (first (send (eval Line) :endpoints)) (volume-points (eval Volume)))
   (pushnew (second (send (eval Line) :endpoints)) (volume-points (eval
Volume))))
```

106

```lisp
(defun make-new-dividing-lines (Volume List-new-points List-old-points)
    (loop for P1 in List-new-points ; handle case where no pre-exitant points
involved
    do (loop for P2 in List-new-points
           do (cond ((not (equal P1 P2))
                    (new-valid-line P1 P2 Volume)))))
    (loop for P1 in List-old-points    ; add pre-exitant lines and points
        do (loop for P2 in List-old-points   ;to new volumes
          do (cond ((not (equal P1 P2))
                    (new-valid-line P1 P2 Volume) ;make new connecting lines
                                              ;then find old ones
                   (loop for Line in (volume-edges (eval Volume))
                      do (let ((endpoint1 (first (send (eval Line) :endpoints)))
                               (endpoint2 (second (send (eval Line) :endpoints))))
                         (cond ((and (or (equal P1 endpoint1)
                                         (equal P1 endpoint2))
                                     (or (equal P2 endpoint1)
                                         (equal P2 endpoint2)))
                               (push-endpoints Line *above*)
                               (push-endpoints Line *below*)
                               (pushnew Line (volume-edges *above*))
                                (pushnew Line (volume-edges *below*))
                               (pushnew Line *lines-in-intercept-plane*)))))))))

    (loop for P-new in List-new-points    ; add new lines connecting old and new
    do (loop for P-old in List-old-points ; points to new volumes
          do (new-valid-line P-new P-old Volume))))

(defun new-valid-line
          (P1 P2 Volume)        ; make a new (and valid) line between P1 and P2
                                ; which lies inside (or along an edge) of Volume
  (loop for F1 in (volume-facets (eval Volume))
     do (cond ((in-facet-p P1 P2 F1)
            (let ((Line (make-line P1 P2)))
            (cond ((not (outside-volume Line Volume))
                  (push-endpoints Line *above*)
                   (push-endpoints Line *below*)
                    (pushnew Line (volume-edges *above*))
                    (pushnew Line (volume-edges *below*))
                  (pushnew Line *lines-in-intercept-plane*)))))))))

(defun simple-volume-test-p ()
  (cond ((or (or (> '3 (length (volume-points *above*)))
             (> '3 (length (volume-points *below*))))
         (or (> '5 (length (volume-edges *above*)))
             (> '5 (length (volume-edges *below*)))))
    (princ "nil (late 1)")
    (return-from simple-volume-test-p 'nil))
   't)

(defun check-eulers-relation-p ()
  (cond ((or (not (equal '2 (+ (length (volume-points *above*))
                     (length (volume-facets *above*))
                  (- '0 (length (volume-edges *above*))))))
         (not (equal '2 (+ (length (volume-points *below*))
                     (length (volume-facets *below*))
                  (- '0 (length (volume-edges *below*)))))))
    (princ " Violated Eulers relation ") (prin1 *precision*)
    (terpri) ;(dump-volumes (list '*above* '*below*))
    (princ "                                                    ")
    (return-from check-eulers-relation-p 'nil)))
   't)
```

```lisp
(defun make-facets (planes volume)
  (loop for P in planes                    ; clear plane properties
    do (setf (get P 'edges) 'nil))

  (loop for P in planes                    ; find which lines lie in which planes
    do (loop for E in (volume-edges (eval Volume))
    do (cond ((subs-line-into-plane-equation E P)
             (setf (get P 'edges) (adjoin E (get P 'edges)))))))))

  (loop for P in planes                    ; build legitimate facets
    do (cond ((and (not (null (get P 'edges)))
             (<= '3 (length (get P 'edges))))
         (setf (volume-facets (eval Volume))
             (adjoin (init-facet-2 (list (get P 'edges) P))
                 (volume-facets (eval Volume)))))))))

  (loop for P in planes                    ; clear plane properties
    do (setf (get P 'edges) 'nil)))


(defun force-facet (Plane)     ; force a facet to exist, if all else fails
  (let* ((lines-in-facet *lines-in-intercept-plane*)
     (forced-facet (init-facet-2 (list lines-in-facet (init-plane Plane)))))
    (setf (volume-facets *above*) (adjoin forced-facet (volume-facets *above*)))
    (setf (volume-facets *below*) (adjoin forced-facet (volume-facets *below*)))
    (princ " Forced ")))


(defun in-facet-p (P1 P2 F)   ; return T iff points P1 and P2 are inside facet F

  (cond ((and (or (member-p (get P1 'lines) (facet-edges (eval F)))
             (member-p P1 (send (eval F) :points)))
         (or (member-p (get P2 'lines) (facet-edges (eval F)))
             (member-p P2 (send (eval F) :points))))
     (return-from in-facet-p 't))
    (t (return-from in-facet-p 'nil))))


(defun outside-volume (Line Volume)   ; return T iff line is outside the volume
                                      ; do only if dealing with ground volumes or
                                      ; non-convex air volumes
  (cond ((or *not-convex-volumes*
         (equal 'ground (volume-composition (eval volume))))
     (let ((mid-point (init-point (send (eval line) :midpoint))))
       (cond ((point-in-volume-p mid-point volume)
             (return-from outside-volume 'nil))
           (t (return-from outside-volume 't)))))
    (t (return-from outside-volume 'nil))))
```

```
;
;    rotation and translation code   cs4452   17may88
;

(defun transpose (A)
  (cond ((null (cdr A)) (mapcar 'list (car A)))
    (t (mapcar 'cons (car A) (transpose (cdr A))))))
(defun dot-product (x y)           ;A vector is a list of numerical atoms.
  (apply '+ (mapcar '* x y)))      ;A matrix is a list of vectors representing
(defun cross-product (x y)
  (list (- (* (cadr x) (caddr y)) (* (caddr x) (cadr y)))
    (- (* (caddr x) (car y)) (* (car x) (caddr y)))
    (- (* (car x) (cadr y)) (* (cadr x) (car y)))))
(defun post-multiply (M x)         ;the rows of the matrix.
  (cond ((null (cdr M)) (list (dot-product (car M) x)))
    (t (cons (dot-product (car M) x) (post-multiply (cdr M) x)))))
(defun pre-multiply (x M)
  (post-multiply (transpose M) x))
(defun matrix-multiply (A B)
  (cond ((null (cdr A)) (list (pre-multiply (car A) B)))
    (t (cons (pre-multiply (car A) B) (matrix-multiply (cdr A) B)))))

(defun cycle-left (L) (mapcar 'row-cycle-left L))
(defun row-cycle-left (R) (append (cdr R) (list (car R))))
(defun cycle-up (M) (append (cdr M) (list (car M))))
(defun unit-vector (one-column length)
  (do ((n length (1- n))
       (R nil (cons (cond ((= one-column n) 1) (t 0)) R)))
      ((zerop n) R)))

(defun concat-matrix (A B)
  (cond ((null A) B)
    (t (cons (append (car A) (car B)) (concat-matrix (cdr A) (cdr B))))))
(defun augment (A) (concat-matrix A (unit-matrix (length A))))
(defun normalize-row (R) (scalar-multiply (/ 1.0 (car R)) R))
(defun scalar-multiply (a x)
  (cond ((null x) nil)
    (t (cons (* a (car x)) (scalar-multiply a (cdr x))))))
(defun solve-first-column (M)
  (do* ((L1 M (cdr L1))
        (L2 (normalize-row (car M)))
        (L3 (list L2) (cons (vector-add (car L1)
                                        (scalar-multiply (- (caar L1)) L2)) L3)))
       ((null (cdr L1)) (reverse L3))))
(defun vector-add (x y) (mapcar '+ x y))
(defun first-n (n R)
  (cond ((zerop n) nil)
    (t (cons (car R) (first-n (1- n) (cdr R))))))
(defun square-car (M)
  (do ((m (length M))
       (L1 M (cdr L1))
       (L2 nil (cons (first-n m (car L1)) L2)))
      ((null L1) (reverse L2))))
(setq A '((1 1 1) (2 1 2) (3 2 3)))
(setq B '((1 1 2) (1 2 3) (2 3 1)))
(defun ncdr (n L) (cond ((zerop n) L) (t (cdr (ncdr (1- n) L)))))
(defun ncar (n L) (cond ((zerop n) nil)
                    (t (cons (car L) (ncar (1- n) (cdr L))))))
(defun nmax-car-first (n L)
  (append (max-car-first (ncar n L)) (ncdr n L)))
```

```lisp
(defun matrix-inverse (M)
  (do ((M1 (max-car-first (augment M))
          (cond ((null M1) nil)
                (t (nmax-car-first n (cycle-left (cycle-up M1))))))
       (n (1- (length M)) (1- n)))
      ((or (minusp n) (null M1)) (cond ((null M1) nil) (t (square-car M1))))
      (setq M1 (cond ((zerop (caar M1)) nil) (t (solve-first-column M1))))))
(defun max-car-first (L)
  (cond ((null (cdr L)) L)
        (t (if (> (abs (caar L)) (abs (caar (max-car-first (cdr L))))) L
               (append (max-car-first (cdr L)) (list (car L)))))))

(defun dh-matrix (cosrotate sinrotate costwist sintwist length translate)
  (list (list cosrotate (- (* costwist sinrotate))
              (* sintwist sinrotate) (* length cosrotate))
        (list sinrotate (* costwist cosrotate)
              (- (* sintwist cosrotate)) (* length sinrotate))
        (list 0. sintwist costwist translate) (list 0. 0. 0. 1.)))


(defun homogeneous-transform (azimuth elevation roll x y z)
  (rotation-and-translation (sin azimuth) (cos azimuth) (sin elevation)
                            (cos elevation) (sin roll) (cos roll) x y z))


(defun rotation-and-translation (spsi cpsi sth cth sphi cphi x y z)
  (list (list (* cpsi cth) (- (* cpsi sth sphi) (* spsi cphi))
              (+ (* cpsi sth cphi) (* spsi sphi)) x)
        (list (* spsi cth) (+ (* cpsi cphi) (* spsi sth sphi))
              (- (* spsi sth cphi) (* cpsi sphi)) y)
        (list (- sth) (* cth sphi) (* cth cphi) z)
        (list 0. 0. 0. 1.)))

(defun A01 (d1)
  (dh-matrix 0 1 0 1 0 d1))
(defun A12 (d2)
  (dh-matrix 0 1 0 1 0 d2))
(defun A23 (d3)
  (dh-matrix 0 1 0 1 0 d3))
(defmacro A03 (d1 d2 d3)
  '(chain-multiply '((A01 ,d1) (A12 ,d2) (A23 ,d3))))
(defun A34 (theta4)
  (dh-matrix (cos theta4) (sin theta4) 0 1 0 0))
(defun A45 (theta5)
  (dh-matrix (cos theta5) (sin theta5) 0 1 0 0))
(defun A56 (theta6)
  (dh-matrix (cos theta6) (sin theta6) 0 1 0 0))
(defmacro A36 (theta4 theta5 theta6)
  '(chain-multiply '((A34 ,theta4) (A45 ,theta5) (A56 ,theta6))))
(defun A06 (d1 d2 d3 theta4 theta5 theta6)
  (matrix-multiply (A03 d1 d2 d3) (A36 theta4 theta5 theta6)))
(setq A6body '((0 0 1 0) (1 0 0 0) (0 1 0 0) (0 0 0 1)))
(defun homogeneous-transform1 (azimuth elevation roll x y z)
  (matrix-multiply (A06 z x y (+ azimuth pi) (- elevation (/ pi 2))
                            (+ roll pi)) A6body))
(setq B6body '((1 0 0 0) (0 0 -1 0) (0 1 0 0) (0 0 0 1)))
(defun homogeneous-transform2 (azimuth elevation roll x y z)
  (matrix-multiply (A06 z x y azimuth elevation roll ) B6body ))

; changes: D.H.Lewis 17 May 88
```

```
(defun unit-matrix (L)
  (loop for i from L downto 1
    collect (loop for j from L downto 1
                  when (equal i j)
                    collect 1
                  else collect 0
            finally)
  finally))

(defun chain-multiply (L)
  (cond ((equal (length L) 2) (matrix-multiply (eval (first L)) (eval (second
L))))
    (t (setq temp (matrix-multiply (eval (first L)) (eval (second L))))
       (chain-multiply (push 'temp (cddr L))))))
```

```
;;; -*- Mode:Common-Lisp; Base:10 -*-
;***********************************************************************************
;
; PATH-DATA                          L.R. WRENN                        31 May 89
;
;
;
;
;  The data for a path is printed out in jet-card form.
;
;***********************************************************************************
;;;
;;;                              FUNCTION LIST
;;;
;;; MAIN:  PATH-DATA   :prints out jetcard type information about a path
;;;        PATH-FOR-IRIS :sends the information needed to display a path
;;;                          on the IRIS to a file of "pathname.dat"
;;;
;;;
;;; SUPPORT FUNCTIONS :
;;;
;;;               distance-XY
;;;               real-to-integer
;;;               convert-number-to-string
;;;               convert-string-to-integer
;;;               find-period-index
;;;               get-leftside-of-real
;;;               get-rightside-of-real
;;;               convert-string-to-real
;;;
;;;***********************************************************************************
(defun path-data (path)              ;prints a jet-card and outputs the
                                     ; total cost of a path
  (let* ((point-list (path-points (eval path)))
    (min-PD-cost '1000000)
    (max-PD-cost '0))
    (terpri)
    (princ "              Leg   Total  Leg   Total  Leg   Fuel   Vol    PD
Leg")
    (terpri)
    (princ "Point         Time  Time   Dist  Dist   Fuel  Remain PD     Cost
Cost")
    (terpri)
    (princ (send (eval (car point-list)) :list-format-real))(terpri)
    (princ "              0.0    0.0    0.0   0.0    0.0   1500   -      -
-")
    (terpri)
    (do* ((start-point  (car point-list)(car point-list))
      (point-list (cdr point-list)(cdr point-list))
      (volume-list (path-volumes (eval path)) (cdr volume-list))
      (next-point (car point-list) (car point-list))
      (leg-dist (distance-XY start-point next-point)
              (distance-XY start-point next-point))
      (total-dist leg-dist (+ leg-dist total-dist))
      (leg-time (/ (distance start-point next-point) (/ 450 60))
              (/ (distance start-point next-point) (/ 450 60))))
      (total-time leg-time (+ leg-time total-time)))
```

112

```lisp
(leg-fuel (fuel-burned (distance start-point
                                 next-point)
                       (climb-angle start-point
                                    next-point)
                       '1500
                       '450)
          (fuel-burned (distance start-point
                                 next-point)
                       (climb-angle start-point
                                    next-point)
                       Fuel-remaining
                       '450))
(fuel-remaining (- 1500 leg-fuel)(- fuel-remaining leg-fuel))
(PD (volume-probability-of-detection (eval (car volume-list)))
    (volume-probability-of-detection (eval (car volume-list))))
(PD-cost (* 100 PD leg-time)(* 100 PD leg-time))
(leg-cost (+ leg-fuel PD-cost) (+ leg-fuel PD-cost))
(total-cost leg-cost (+ leg-cost total-cost))
(min-PD-cost (cond((< PD-cost min-PD-cost)
                   PD-cost)
                  (t min-PD-cost))
             (cond((< PD-cost min-PD-cost)
                   PD-cost)
                  (t min-PD-cost)))
(max-PD-cost (cond((> PD-cost max-PD-cost)
                   PD-cost)
                  (t max-PD-cost))
             (cond((> PD-cost max-PD-cost)
                   PD-cost)
                  (t max-PD-cost))))
((null (second point-list))
(princ (send (eval next-point) :list-format-real))(terpri)(princ "
")
(princ (format nil "~7,1F" leg-time))
(princ (format nil "~7,1F"  total-time))
(princ (format nil "~7,1F"  leg-dist))
(princ (format nil "~7,1F"  total-dist))
(princ (format nil "~7,1F"  leg-fuel))
(princ (format nil "~7,1F"  fuel-remaining))
(princ (format nil "~6,3F"  PD))
(princ (format nil "~7,1F"  PD-cost))
(princ (format nil "~7,1F"  leg-cost))(terpri)
(princ "Total cost of this path - ")
(princ (format nil "~7,1F"  total-cost))(terpri)
(princ "minimum PD cost - ")
(princ (format nil "~7,1F"  min-PD-cost))(terpri)
(princ "maximum PD cost - ")
(princ (format nil "~7,1F"  max-PD-cost))(terpri)
(princ "average PD cost - ")
(princ (format nil "~7,1F"  (/ total-cost total-time)))(terpri)
(terpri) total-cost)
(princ  (send (eval next-point) :list-format-real))(terpri)(princ "      ")
(princ (format nil "~7,1F" leg-time))
(princ (format nil "~7,1F"  total-time))
(princ (format nil "~7,1F"  leg-dist))
(princ (format nil "~7,1F"  total-dist))
(princ (format nil "~7,1F"  leg-fuel))
(princ (format nil "~7,1F"  fuel-remaining))
(princ (format nil "~6,3F"  PD))
(princ (format nil "~7,1F"  PD-cost))
(princ (format nil "~7,1F"  leg-cost))
(terpri))))
```

113

```
(defun distance-XY (pt1 pt2)      ;finds the ground distance
                                  ; between two points
  (sqrt (+ (* (- (point-X-coord (eval pt1)) (point-X-coord (eval pt2)))
              (- (point-X-coord (eval pt1)) (point-X-coord (eval pt2))))
           (* (- (point-y-coord (eval pt1)) (point-y-coord (eval pt2)))
              (- (point-y-coord (eval pt1)) (point-y-coord (eval pt2)))))))


(defun path-for-IRIS (path)     ;makes a file of the points of a path for
                                ; use on IRIS Graphic Display
  (setq *output-stream* (open (string-append "exp3:wrennthesis;"
                                    (symbol-name path)
                                    ".dat") :direction :output))
  (print "sending path data to file ('path-name'.dat)...")
  (let* ((point-list (path-points (eval path)))
         (start-point-list  (send (eval (car point-list)) :list-format-real))
         (list-length (length point-list))
         (volume-list (path-volumes (eval path))))
    (terpri)
    (princ list-length *output-stream*)(terpri *output-stream*)
    (princ (format nil "~8,2F"  (first start-point-list)) *output-stream*)
    (princ (format nil "~8,2F"  (third start-point-list)) *output-stream*)
    (princ (format nil "~8,2F"  (* -1.0 (second start-point-list)))
*output-stream*)
    (princ (format nil "~7,3F" (volume-probability-of-detection
                        (eval (car volume-list)))) *output-stream*)
    (terpri *output-stream*)

    (do* ((point-list (cdr point-list)(cdr point-list))
      (start-point-list  (send (eval (car point-list)) :list-format-real)
                         (send (eval (car point-list)) :list-format-real))

      (volume-list (path-volumes (eval path)) (cdr volume-list))
      (PD (volume-probability-of-detection (eval (car volume-list)))
          (volume-probability-of-detection (eval (car volume-list)))))
      ((null (second point-list))
       (princ (format nil "~8,2F"  (first start-point-list)) *output-stream*)
       (princ (format nil "~8,2F"  (third start-point-list)) *output-stream*)
       (princ (format nil "~8,2F"
                    (* -1.0 (second start-point-list))) *output-stream*)
       (princ (format nil "~7,3F"
                    (volume-probability-of-detection
                       (eval (car volume-list)))) *output-stream*)
       (terpri *output-stream*) PD)
      (princ (format nil "~8,2F"  (first start-point-list)) *output-stream*)
      (princ (format nil "~8,2F"  (third start-point-list)) *output-stream*)
      (princ (format nil "~8,2F"
              (* -1.0 (second start-point-list)) *output-stream*)
      (princ (format nil "~7,3F" (volume-probability-of-detection
                        (eval (car volume-list)))) *output-stream*)
      (terpri *output-stream*))

    )
  (close *output-stream*)
  (print "Done.")
  'nil)


(defun real-to-integer (realnum)   ;returns integer part of real number
  (get-leftside-of-real (convert-number-to-string realnum)))
```

114

```lisp
(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)))
```

```
;;; -*- Mode:Common-Lisp; Base:10 -*-
;**********************************************************************
;
; PATH-OPTIMIZATION              L.R. WRENN                  6 Mar 89
;
;
;
;
;  The optimization code optimizes the initial A* path according
; to snells law criteria.
;
;**********************************************************************
;
;    THESIS                              L.R.WRENN    15 JUNE 1989
;
; MAIN FUNCTIONS:  RANDOM-RAY-OPTIMIZE
;                  RANDOM-RAY-OPT2
;                  REVISE-PATH
;
; SUPPORT FUNCTIONS:
;                  ADJUST-PATH-INTO-LAST-VOLUME
;                  ADJUST-PATH-INTO-LAST-VOLUME-2
;                  REFINE-LINE-TO-GOAL
;                  ADJUST-VECTOR
;                  REVISE-PATH-2
;                  CONNECT-POINTS
;                  GET-VECTOR-AND-FACTOR
;                  SWITCH-ADJUSTMENTS
;                  ADJUST-POINT
;                  NORMAL-LINE
;                  MAKE-A PATH-PLANE
;                  ANGLE-BETWEEN-LINE-FACETN
;                  ANGLE-BETWEEN-LINES-SMALLEST
;                  FIND-SNELLS-ANGLE
;                  FIND-OUTBOUND-VECTOR
;                  FIND-OUTBOUND-VECTOR-2
;                  FIND-OUTBOUND-VECTOR-3
;                  SOLVE-QUADRATIC
;                  FIND-OUTBOUND-LINE-2
;                  FIND-POINT-FROM-COEF-AND-POINT
;                  MAKE-UNIT-LINE
;                  PARALLEL-LINES
;                  FINE-INTERCEPT-POINT-EXTENDED
;                  GET-INTERCEPT-POINT-2-EXTENDED
;                  NORMALIZE-VECTOR
;                  GET-ADGUSTMENT-VECTORS
;                  CHECK-FACET-LIST-AGAINST-SNELLS-LAW
;
;**********************************************************************
(defvar *reflectance*)
```

116

```lisp
(defun random-ray-optimize (path-list)    ;Takes a list of paths are trys
                                          ; random ray optimization on them
                                          ; returns a list of paths that worked
                                          ; or nil corresponding to those that
                                          ; did not.
                             ; ex. (random-ray-optimization '(|path0006|
|path0011|))
  (let ((new-paths ))
    (do* ((old-path-list path-list (cdr old-path-list))
       (current-path (car old-path-list)(car old-path-list))
       (random-ray-worked (random-ray-opt2 current-path)
                          (random-ray-opt2 current-path))

       (new-path-list (cond ((null random-ray-worked)
                               (list nil))
                             (t (list (revise-path current-path
                                               random-ray-worked))))
                   (cond ((null random-ray-worked)
                           (cons  'nil new-path-list))
                         (t (cons (revise-path current-path
                                           random-ray-worked)
                               new-path-list)))))
       ((null (cdr old-path-list))
        (setf new-paths (reverse new-path-list))))

    (terpri)
    (princ "Old Paths  -  New Paths")(terpri)
    (do ((old-path (car path-list)(car path-list))
       (new-path (car new-paths)(car new-paths))
       (path-list (cdr path-list)(cdr path-list))
       (new-paths (cdr new-paths)(cdr new-paths)))
      ((null path-list)(princ old-path)(princ "   -   ")
       (princ new-path)(terpri) new-paths)
        (princ old-path)(princ "   -   ")(princ new-path)(terpri))
    new-paths))
```

```lisp
(defun random-ray-opt2 (path)          ; Takes a path and checks to see if it is
                                       ; possible to pass a random ray through
                                       ; the volumes obeying Snell's Law at all
                                       ; facets.  Will return a line if it can
                                       ; or 'nil if it cannot.
                                       ; ex. (random-ray-opt2 '|path0006|)
  (setf *reflectance* 5)
  (let* ((line-to-goal (make-line (path-start-point (eval path))
                             (path-end-point (eval path))))
     (facet-list (path-facets (eval path)))
     (volume-list (path-volumes (eval path))))
    (do* ((IP
        (find-intercept-point-extended (car facet-list)
                                 line-to-goal)
         (find-intercept-point-extended (car facet-list)
                                 line-to-goal)))
      ((point-in-volume-P IP (car volume-list))
       (setf line-to-goal (make-line
                         (path-start-point (eval path))
                         (find-intercept-point-extended (car facet-list)
                                                 line-to-goal))))
       (setf line-to-goal (make-line (path-start-point (eval path))
                             (init-point
                               (average-points
                                IP
                                (facet-center
                                 (eval (car facet-list)))))))))
    (cond ((not (null (check-facet-list-against-snells-law
                   line-to-goal facet-list volume-list)))
       (terpri)(princ "A random solution has been found into the goal volume")
       (terpri)(princ "The line to start the path is - ")
       (princ line-to-goal)(terpri)
       (refine-line-to-goal line-to-goal path))
      (t
       (setf line-to-goal (adjust-path-into-last-volume
                         line-to-goal facet-list volume-list))
       (cond ((null line-to-goal)
             (terpri)
             (princ "There is no solution to the random ray optimization")
             (terpri)(princ "Try one of the other optimizations")(terpri)
             (return-from random-ray-opt2 'nil)))
       (terpri)
       (princ "A random solution has been found into the goal volume")
       (terpri)
       (princ " by adjusting the line to goal. The line to start the path is -
")
       (princ line-to-goal)(terpri)
       (princ "the line in the last volume is - ")
       (princ (check-facet-list-against-snells-law
               line-to-goal facet-list volume-list))(terpri)
       (refine-line-to-goal line-to-goal path))
      )))
```

118

```lisp
(defun adjust-path-into-last-volume              ;This function is called
        (line-to-goal facet-list volume-list)    ; recurisively to find an
                                                 ; adjustable line to the final
                                                 ; volume of a path.  Returns
                                                 ; the line or 'nil
  (let* ((line-out-of-last-facet)
     (IP)
     (adjustment-vectors)
     (miss-distance 999999.0))
     (cond ((not(null (cdr facet-list)))
        (setf line-to-goal (adjust-path-into-last-volume
                        line-to-goal
                        (but-last facet-list)
                        (but-last volume-list)))))
     (terpri)(princ "In check-line-with-adjustments")(terpri)
     (princ "facets - ")(princ facet-list)(terpri)
     (princ "volumes - ")(princ volume-list)(terpri)
     (princ "line-to-goal - ")(princ line-to-goal)(terpri)

     (cond ((null line-to-goal)
       (return-from adjust-path-into-last-volume 'nil)))

     (cond ((null (cdr facet-list))
       (setf line-out-of-last-facet line-to-goal))
      (t(setf line-out-of-last-facet (check-facet-list-against-snells-law
                           line-to-goal
                           (but-last facet-list)
                           (but-last volume-list)))))


     (cond ((null line-out-of-last-facet)
       (return-from adjust-path-into-last-volume 'nil)))

     (setf IP (find-intercept-point-extended (car (last facet-list))
                              line-out-of-last-facet))
     (princ "The intercept point is - ")(princ IP)(terpri)
     (cond ((and (point-in-volume-P IP (car (last volume-list)))
          (check-facet-list-against-snells-law
            line-to-goal
            facet-list
            volume-list))
       (return-from adjust-path-into-last-volume line-to-goal))
      (t (setf adjustment-vectors (get-adjustment-vectors
                         IP
                         (car (last facet-list))))
       (setf miss-distance (distance
                    IP
                    (facet-center
                      (eval (car (last facet-list))))))))))

     (princ "The adjustment-vectors and miss-distance is - ")(terpri)
     (princ adjustment-vectors)(terpri)(princ miss-distance)(terpri)

     (setf line-to-goal (adjust-path-into-last-volume-2
                    line-to-goal
                    IP
                    miss-distance
                    facet-list
                    volume-list
                    adjustment-vectors
                    line-out-of-last-facet))
  line-to-goal))
```

119

```lisp
(defun adjust-path-into-last-volume-2 (line-to-goal
                         IP
                         miss-distance
                         facet-list
                         volume-list
                         adjustment-vectors
                         line-out-of-last-facet)
  ;this is the actual section that does the adjustments
  ;This do* will be exited with a valid new line-to-goal and
  ; line-out-of-last-facet or will exit with 'nil causing no path to be found

 (do* ((new-line-to-goal line-to-goal)
  (adjustment-factor '125)
  (adjust-temp)
  (IP-2 IP)
  (old-reflection 'nil)
  (IP-90deg 'nil)
  (reflected 'nil)
  (new-miss-distance miss-distance)
  (adjustment-list '("in" "down" "out" "up" "change")
                (cond (reflected
                       (cond (((<= old-reflection *reflectance*)
                              (setf adjust-temp
                                   (switch-adjustments
                                     adjustment-list adjustment-factor))
                              (setf adjustment-factor (cadr adjust-temp))
                              (car adjust-temp))
                             (t (setf old-reflection *reflectance*)
                               adjustment-list)))

                      ((< miss-distance new-miss-distance)
                      (setf adjust-temp
                            (switch-adjustments
                             adjustment-list adjustment-factor))
                      (setf adjustment-factor (cadr adjust-temp))
                      (car adjust-temp))
                      (t
                      (setf miss-distance new-miss-distance)
                      adjustment-list))))

          ;exit condition
      ((and (point-in-volume-P IP (car (last volume-list)))
        (not (null(check-facet-list-against-snells-law
                 new-line-to-goal
                 facet-list
                 volume-list)))))

  (setf line-to-goal new-line-to-goal))


  (terpri)

  (cond((< adjustment-factor '1)
    (return-from adjust-path-into-last-volume-2 'nil)))
```

120

```lisp
(cond ((and(null (but-last facet-list))    ;is there only one facet and
                                           ;is IP on it
          (point-in-volume-P IP (car (last volume-list))))
    (princ "adjustment hit facet but reflected in first volume ")(terpri)
    (cond (reflected
          (setf reflected 't)
          (setf adjustment-list '("in" "down" "out" "up" "change"))
          (setf adjustment-factor '125)))
     (cond((null old-reflection)
          (setf old-reflection *reflectance*)))
    (princ "*reflectance* - ")(princ *reflectance*)(terpri)
    (princ "old-reflection - ")(princ old-reflection)(terpri)
    (cond((< *reflectance* old-reflection)
          (setf line-to-goal new-line-to-goal)
          (setf IP IP-2))))


  ((point-in-volume-P IP (car (last volume-list)))
   (terpri)
   (princ "We have an intersect point but it will not go through")(terpri)
   (cond (reflected
         (setf reflected 't)
         (setf adjustment-list '("in" "down" "out" "up" "change"))
         (setf adjustment-factor '125)))
   (cond((null old-reflection)
         (setf old-reflection *reflectance*)))

   (setf IP-90deg
        (find-intercept-point-extended
          (car(last facet-list))
          (make-unit-line (send (eval line-out-of-last-facet)
                             :start-point)
                        (normal-line IP
                                  (car (last facet-list)))))))
    (setf miss-distance (distance IP
                            IP-90deg))
           ))


(setf new-line-to-goal (make-line
                  (vector-start-point
                    (eval (line-segment-direction-vector
                            (eval line-to-goal))))
                  (adjust-point
                    (vector-end-point
                     (eval (line-segment-direction-vector
                            (eval line-to-goal))))
                    (car (get-vector-and-factor
                        adjustment-vectors
                        adjustment-list
                        adjustment-factor))
                    (cadr (get-vector-and-factor
                        adjustment-vectors
                        adjustment-list
                        adjustment-factor))))))
```

```
      (cond((null (but-last facet-list))
        (princ "adjustment missed everything out of first facet adjustments made
")
      (terpri)
      (setf new-line-to-goal (make-line (send (eval line-to-goal)
                                        :start-point)
                                   (facet-center
                                     (eval (car (last facet-list)))))))
      (setf IP-2 (find-intercept-point-extended (car (last facet-list))
                                        new-line-to-goal))
      (setf new-miss-distance (distance
                              IP-2
                              (facet-center
                                (eval(car facet-list)))))
      (cond((and (null reflected)(< new-miss-distance miss-distance))
          (setf line-to-goal new-line-to-goal)
          (setf IP IP-2))))


                 ;we missed the last facet-see if we
                      ;missed the next to the last
    ((null (check-facet-list-against-snells-law
          new-line-to-goal
          (but-last facet-list)
          (but-last volume-list)))
     (princ "adjustment missed everything ")(terpri)

     (setf new-miss-distance (+ 1 miss-distance)))


    (t (setf line-out-of-last-facet (check-facet-list-against-snells-law
                                  new-line-to-goal
                                  (but-last facet-list)
                                  (but-last volume-list)))
      (princ "adjustment may be ok")(terpri)
      (setf IP-2 (find-intercept-point-extended (car (last facet-list))
                                        line-out-of-last-facet))
      (setf new-miss-distance (distance
                              IP-2
                              (cond (reflected IP-90deg)
                                    (t (facet-center
                                         (eval(car
                                            (last facet-list)))))))))

      (cond((and (null reflected) (< new-miss-distance miss-distance))
          (setf line-to-goal new-line-to-goal)
          (setf IP IP-2)))))
  )
line-to-goal)
```

```lisp
(defun refine-line-to-goal (line path) ;this function will adjust the
                                       ; line as close to the actual
                                       ; goal as it can and report the results
                                       ; Returns the best line
  (terpri)(princ "In refine-line-to-goal ")
  (terpri)(princ "The path we are optimizing is - ")(princ path)(terpri)
  (let* ((facet-list (path-facets (eval path)))
         (volume-list (path-volumes (eval path')))
         (start-point (path-start-point (eval path)))
         (line-to-goal line)
         (adjustment-vector)
         (adjustment-factor '125)
         (angle pi)
         (check-line (check-facet-list-against-snells-law
                        line-to-goal
                        facet-list
                        volume-list)))
    (princ "check-line looks like - ")(princ check-line)(terpri)
      (do* ((line-out-of-last-facet (check-facet-list-against-snells-law
                                       line-to-goal
                                       facet-list
                                       volume-list)
                                     (check-facet-list-against-snells-law
                                       new-line-to-goal
                                       facet-list
                                       volume-list))
            (line-facet-to-goal (make-line
                                   (send (eval line-out-of-last-facet) :start-point)
                                   (path-end-point (eval path)))
                                 (make-line
                                   (send (eval line-out-of-last-facet) :start-point)
                                   (path-end-point (eval path))))
            (dist-facet-to-goal (send (eval line-facet-to-goal) :length)
                                (distance (send
                                            (eval line-out-of-last-facet)
                                            :start-point)
                                          (path-end-point (eval path))))
            (new-line-to-goal line-to-goal)
            (new-angle (angle-between-lines-smallest line-facet-to-goal
                                       line-out-of-last-facet)
                       (angle-between-lines-smallest line-facet-to-goal
                                       line-out-of-last-facet)))
           ((< new-angle '0.0025) line-to-goal)
        (terpri)
        (princ "point in last volume we are trying to adjust - ")
        (princ (send
                 (eval (adjust-point
                         (send (eval line-out-of-last-facet) :start-point)
                         (normalize-vector(send
                                            (eval
                                              (line-segment-direction-vector
                                                (eval line-out-of-last-facet)))
                                            :list-format))
                         dist-facet-to-goal))
                 :list-format-real))(terpri)
```

123

```lisp
(setf adjustment-vector
  (list (normalize-vector
          (append
            (send
              (eval (line-segment-direction-vector
                      (eval
                        (make-line
                          (adjust-point
                            (send
                              (eval line-out-of-last-facet)
                              :start-point)
                            (normalize-vector(send
                                  (eval
                                    (line-segment-direction-vector
                                      (eval line-out-of-last-facet)))
                                    :list-format))
                              dist-facet-to-goal)
                            (path-end-point (eval path))
                            ))))
                  :list-format)
                '(0))))))
(princ "the adjustment-vector is - ")(princ adjustment-vector)(terpri)
(print "the adjustment-factor is - ")(print adjustment-factor)(terpri)
(setf new-line-to-goal (make-line
                  start-point
                  ;(vector-start-point
                  ; (eval (line-segment-direction-vector
                  ;   (eval new-line-to-goal))))
                  (adjust-point
                    (vector-end-point
                      (eval (line-segment-direction-vector
                              (eval line-to-goal))))
                    (car (get-vector-and-factor
                          adjustment-vector
                          '("in")
                          adjustment-factor))
                    (cadr (get-vector-and-factor
                          adjustment-vector
                          '("in")
                          adjustment-factor)))))
(princ "check of new line - ")
(princ (check-facet-list-against-snells-law
          new-line-to-goal
          facet-list
          volume-list))(terpri)
(cond ((null (check-facet-list-against-snells-law
          new-line-to-goal
          facet-list
          volume-list))
    (princ "new line did not meet snells law")(terpri)
    (setf adjustment-factor (/ adjustment-factor 5))
    (setf new-line-to-goal line-to-goal))
  (t (cond ((< new-angle angle)
            (setf angle new-angle)
            (setf line-to-goal new-line-to-goal))
          (t (setf new-line-to-goal line-to-goal)
            (setf adjustment-factor (/ adjustment-factor 5))))))
(cond ((< adjustment-factor '0.008)
    (princ "Adjusted as close as possible but still missed goal")(terpri)
    (princ "Miss angle in radians is - ")(princ angle)(terpri)
    (return-from refine-line-to-goal new-line-to-goal))))
line))
```

```
(defun revise-path (path line)        ;Takes the old path and the new random ray
                                      ; line and makes a new path out of then.
                                      ; Returns new path name
  (let ((line-list)
    (new-path)
    (path-list (revise-path-2 line
                             (path-facets (eval path))
                             (path-volumes (eval path))
                             (list (path-start-point (eval path)))))))
    (setf path-list (reverse (cons (path-end-point (eval path))
                             path-list)))

    (setf line-list (connect-points path-list))
    (setf new-path (init-new-path (path-start-point (eval path))
                             (path-end-point (eval path))
                             (path-volumes (eval path))
                             (path-facets (eval path))
                             line-list
                             path-list
                             'nil
                             'nil))
    (calc-path-and-stats new-path)
    new-path))



(defun revise-path-2                  ;Called recursively to revise the old
                                      ; path to a goal with the random ray
       (line facet-list volume-list point-list)
  (cond ((not (null (cdr facet-list)))
     (setf point-list (revise-path-2 line
                             (but-last facet-list)
                             (but-last volume-list)
                             point-list))))
    (setf point-list (cons (send (eval (check-facet-list-against-snells-law
                             line
                             facet-list
                             volume-list))
                      :start-point)
                point-list))
  point-list)



(defun connect-points (points-list)   ;Connects a list of points and
                                      ; returns the list of lines
  (do* ((current-point (car points-list) (car new-points-list))
    (new-points-list (cdr points-list) (cdr new-points-list))
    (line-list (list (make-line current-point (car new-points-list)))
            (cons (make-line current-point (car new-points-list))
                line-list)))
      ((null (cdr new-points-list)) (reverse line-list))))



(defun but-last (listL) ;returns all but the last item in the list
  (reverse (cdr (reverse listL)))))
```

```lisp
(defun get-vector-and-factor (adjustment-vectors
                              curr-adj-list
                              curr-adj)
  (let ((return-list))
    (cond ((equal "in" (car curr-adj-list))
           (setf return-list
                 (list (car adjustment-vectors)
                       curr-adj)))
          ((equal "up" (car curr-adj-list))
           (setf return-list
                 (list (cadr adjustment-vectors)
                       curr-adj)))

          ((equal "out" (car curr-adj-list))
           (setf return-list
                 (list (car adjustment-vectors)
                       (* '-1 curr-adj))))
          ((equal "down" (car curr-adj-list))
           (setf return-list
                 (list (cadr adjustment-vectors)
                       (* -1 curr-adj)))))
    return-list))

(defun switch-adjustments (curr-adj-list curr-adj)
  (setf curr-adj-list (append (cdr curr-adj-list)(list (car curr-adj-list))))
  (cond ((equal "change" (car curr-adj-list))
         (setf curr-adj (/ curr-adj 5))
         (setf curr-adj-list (append (cdr curr-adj-list)
                                     (list (car curr-adj-list))))))
  (list curr-adj-list curr-adj))

(defun adjust-point (point vector factor)
  (init-point
    (map 'list '+ (send (eval point) :list-format)
    (scalar-multiply
      factor
      vector))))


(defun normal-line (point facet)      ;makes normal of facet into a line at point
  (let* ((end-point-normal-line
          (init-point (map 'list '+ (send (eval Point) :list-format)
                           (map 'list '* '(100 100 100)
                           (send (eval facet) :list-coeff-3)))))
         (N-line (make-line Point end-point-normal-line)))
    N-line))

(defun make-a-path-plane (Line-1 Facet)   ; makes a plane containing the
                                          ; normal of a plane and some
                                          ; line not in that plane but
                                          ; that intersects it.  If line
                                          ; is perpendicular to the plane
                                          ; it will be a vertical plane.

  (let* ((point-intersect (find-intercept-point facet line-1))
         (line-N (normal-line point-intersect facet))
         (end-point-on-normal-line
           (vector-end-point (eval
                              (line-segment-direction-vector (eval line-N))))))
    (cond ((parallel-lines line-N line-1)
           (return-from make-a-path-plane (make-vertical-plane line-1)))
          (t (make-a-plane end-point-on-normal-line line-1)))))
```

126

```
(defun angle-between-line-facetN (line-1 facet)    ; finds the angle between
                                                   ; line-1 and the normal
                                                   ; of plane, line-1 and plane
                                                   ; must intersect
                                                   ; 0 is perpendicular to plane
  (let* ((point-intersect (find-intercept-point facet line-1))
     (line-N (normal-line point-intersect facet))
     (angle 'nil))
     (cond ((parallel-lines line-N line-1)
       (return-from angle-between-line-facetN '0))
       (t (setf angle (angle-between-lines line-N line-1))))
     (cond ((GT angle *PI2*)
       (setf angle (- *PI* angle))))
     angle))



(defun angle-between-lines-smallest (L1 L2)
  (let ((angle (angle-between-lines L1 L2)))
     (cond ((GT angle *PI2*)
       (setf angle (- *PI* angle))))
     angle))



(defun find-snells-angle (Line-1 Facet Cost-1 Cost-2)
                                   ; Finds outbound snells angle assuming
                                   ;
                                   ; Cost-1 * sin(theta-1) = Cost-2 *
sin(theta-2)

                                   ;
                                   ;  where theta-x is the angle between line
and
                                   ;  the normal to the plane
                                   ; Line-1 MUST intersect Facet
  (let* ((theta-1 (angle-between-line-facetN line-1 facet))
     (theta-2 'nil)
     (temp))
     (cond ((zerop Cost-1)
       (setf cost-1 '.01)))    ; forces going from 0% to 99% to be within
     (cond ((zerop Cost-2)            ; 1/2 a degree on perpendicular to plane
       (setf cost-2 '.01)))    ; ie. shortest path out of volume
     (cond ((zerop theta-1)
       (return-from find-snells-angle '0))
       ((equal Cost-1 Cost-2)
       (return-from find-snells-angle theta-1))
       (t (setf temp (/ (* Cost-1 (sin theta-1)) Cost-2))
         (terpri)(princ temp)(terpri)
         (cond ((> temp '1.0)
             (terpri)
             (setf *reflectance* temp)
             (princ "Reflection inside volume by Snell's Law")
             (terpri)
             (return-from find-snells-angle '"reflect"))
             (t (setf theta-2 (asin temp)))))))
     theta-2))
```

127

```
(defun find-outbound-vector (M)
  (let* ((equ1 (car M))
     (equ2 (cadr M))
     (A12 (first equ1))
     (B12 (second equ1))
     (C12 (third equ1))
     (d1 (first equ2))
     (e1 (second equ2))
     (f1 (third equ2))
     (K0 (fourth equ2))
     (test1 (- (* C12 e1)
            (* B12 f1)))
     (K1)
     (K2)
     (K3)
     (K4)
     (quad-equ)
     (d21)
     (d22))
    (cond ((or (< (abs test1) '0.00001)
        (zerop C12))(princ "aborted process - division by zero")
      (terpri)(princ "Trying find-outbound-vector-2")(terpri)
      (return-from find-outbound-vector (find-outbound-vector-2 M))))
    (setf K1 (/ (* K0 C12) test1))
    (setf K2 (/ (- (* A12 f1) (* C12 d1)) test1))
    (setf K3 (/ (* B12 K1) (- C12)))
    (setf K4 (/ (+ A12 (* B12 K2)) (- C12)))
    (setf quad-equ (list (+ 1 (* K2 K2) (* K4 K4))
             (+ (* 2 K1 K2) (* 2 K3 K4))
             (+ -1 (* K1 K1) (* K3 K3))))
    (setf d21 (car (solve-quadratic quad-equ)))
    (setf d22 (cadr (solve-quadratic quad-equ)))

    (cond ((complexp d21)(princ "aborted process - complex numbers")
      (terpri)(princ "Trying find-outbound-vector-2")(terpri)
      (return-from find-outbound-vector (find-outbound-vector-2 M))))

    (list (cond ((complexp d21)(list nil))
        (t (list d21 (+ K1 (* K2 d21)) (+ K3 (* K4 d21)))))
      (cond ((complexp d22)(list nil))
        (t (list d22 (+ K1 (* K2 d22)) (+ K3 (* K4 d22))))))
))


(defun find-outbound-vector-2 (M)
  (let* ((equ1 (car M))
     (equ2 (cadr M))
     (A12 (first equ1))
     (B12 (second equ1))
     (C12 (third equ1))
     (d1 (first equ2))
     (e1 (second equ2))
     (f1 (third equ2))
     (K0 (fourth equ2))
     (test1 (- (* B12 d1)
            (* A12 e1)))
     (K1)
     (K2)
     (K3)
     (K4)
     (quad-equ)
     (f21)
```

128

```lisp
     (f22))
    (cond ((or (< (abs test1) '0.00001)
           (zerop B12))(princ "aborted process - division by zero")
      (terpri)(princ "Trying find-outbound-vector-3")(terpri)
      (return-from find-outbound-vector-2 (find-outbound-vector-3 M)))))
    (setf K1 (/ (* K0 B12) test1))
    (setf K2 (/ (- (* C12 e1) (* B12 f1)) test1))
    (setf K3 (/ (* A12 K1) (- B12)))
    (setf K4 (/ (+ C12 (* A12 K2)) (- B12)))
    (setf quad-equ (list (+ 1 (* K2 K2) (* K4 K4))
                    (+ (* 2 K1 K2) (* 2 K3 K4))
                    (+ -1 (* K1 K1) (* K3 K3))))
    (setf f21 (car (solve-quadratic quad-equ)))
    (setf f22 (cadr (solve-quadratic quad-equ)))

    (cond ((complexp f21)(princ "aborted process - complex numbers")
      (terpri)(princ "Trying find-outbound-vector-3")(terpri)
      (return-from find-outbound-vector-2 (find-outbound-vector-3 M))))

    (list (cond ((complexp f21)(list nil))
          (t (list (+ K1 (* K2 f21)) (+ K3 (* K4 f21)) f21)))
     (cond ((complexp f22)(list nil))
          (t (list (+ K1 (* K2 f22)) (+ K3 (* K4 f22)) f22)))))
))


(defun find-outbound-vector-3 (M)
  (let* ((equ1 (car M))
     (equ2 (cadr M))
     (A12 (first equ1))
     (B12 (second equ1))
     (C12 (third equ1))
     (d1 (first equ2))
     (e1 (second equ2))
     (f1 (third equ2))
     (K0 (fourth equ2))
     (test1 (- (* A12 f1)
            (* C12 d1)))
     (K1)
     (K2)
     (K3)
     (K4)
     (quad-equ)
     (e21)
     (e22))
    (cond ((or (< (abs test1) '0.00001)
           (zerop A12))(princ "aborted process - division by zero")
      (terpri)(princ "Nothing else to try")(terpri)
      (return-from find-outbound-vector-3 '"div-by-zero")))
    (setf K1 (/ (* K0 A12) test1))
    (setf K2 (/ (- (* B12 d1) (* A12 e1)) test1))
    (setf K3 (/ (* C12 K1) (- A12)))
    (setf K4 (/ (+ B12 (* C12 K2)) (- A12)))
    (setf quad-equ (list (+ 1 (* K2 K2) (* K4 K4))
                    (+ (* 2 K1 K2) (* 2 K3 K4))
                    (+ -1 (* K1 K1) (* K3 K3))))
    (setf e21 (car (solve-quadratic quad-equ)))
    (setf e22 (cadr (solve-quadratic quad-equ)))


    (cond ((complexp e21)(princ "aborted process - complex numbers")
```

```lisp
          (terpri)(princ "Nothing else to try")(terpri)
          (return-from find-outbound-vector-3 '((nil)(nil)) )))


     (list (cond ((complexp e21)(list nil))
            (t (list (+ K3 (* K4 e21)) e21 (+ K1 (* K2 e21)))))
        (cond ((complexp e22)(list nil))
            (t (list (+ K3 (* K4 e22)) e22 (+ K1 (* K2 e22))))))
     ))


(defun solve-quadratic (QE)
  (let ((intermediate-sqrt-term (- (* (second QE)(second QE))
                          (* 4 (first QE)(third QE))))
     (sqrterm '0))
    (cond ((and (>= intermediate-sqrt-term '-0.1)
          (< intermediate-sqrt-term '0))
; (terpri)
; (princ "**** SQUARE ROOT OF SMALL NEGATIVE NUMBER ABOUT TO BE TAKEN ****")
; (terpri)
; (princ "**** NUMBER CHANGED TO ZERO TO AVOID COMPLEX NUMBER ****")(terpri)
        (setf intermediate-sqrt-term '0)))
     (setf sqrterm (sqrt intermediate-sqrt-term))
     (list (/ (+ (- (second QE)) sqrterm)(* 2 (first QE)))
       (/ (- (- (second QE)) sqrterm)(* 2 (first QE)))))))


(defun find-outbound-line-2 (Line Facet Cost-1 Cost-2)
                              ; Finds outbound line from a Facet using
                              ; snells law and solving for three equations
                              ; Line-1 MUST intersect Facet
                              ; check to make sure line-1 is
                              ; not perpenduclar to facet
  (let* ((point-intersect (find-intercept-point facet line))
     (line-1 (make-unit-line point-intersect line))
     (theta-in (angle-between-line-facetN line-1 facet))
     (theta-out (find-snells-angle Line Facet Cost-1 Cost-2))
     (path-plane (make-a-path-plane Line Facet))
     (equation-1a (reverse (cons
                  '0
                  (cdr
                    (reverse (send (eval path-plane)
                              :list-coeff))))))
                          ; plane Ax + By + Cz = Ao
     (equation-1 (normalize-vector equation-1a))
     (equation-2)
     (Two-equations)
     (two-vectors)
     (angle-of-new-line-with-normal-1 'nil)
     (angle-of-new-line-with-normal-2 'nil))
    (cond ((equal "reflect" theta-out)
      (return-from find-outbound-line-2 theta-out))
     ((zerop theta-out)
      (return-from find-outbound-line-2
        (make-line
          point-intersect
          (init-point
          (map 'list '+ (send (eval point-intersect) :list-format)
               (scalar-multiply
                10
                (send
                  (eval(line-segment-direction-vector (eval line-1)))
                  :list-format)))))))))))
```

```
(setf equation-2 (append
            (send
             (eval (line-segment-direction-vector (eval line-1)))
             :list-format)
            (list (cos (- theta-in theta-out)))))))
(setf Two-equations (list equation-1 equation-2))

(setf two-vectors (find-outbound-vector Two-equations))

(cond ((null(caar two-vectors))
   (setf angle-of-new-line-with-normal-1 nil)
   (setf angle-of-new-line-with-normal-2 nil))
  (t (setf angle-of-new-line-with-normal-1
         (angle-between-line-facetN
           (make-line
             point-intersect
             (find-point-from-coef-and-point
             point-intersect
             (car two-vectors)))
           facet))
     (setf angle-of-new-line-with-normal-2
         (angle-between-line-facetN
           (make-line
             point-intersect
             (find-point-from-coef-and-point
             point-intersect
             (cadr two-vectors)))
           facet))))
(cond ((and (null angle-of-new-line-with-normal-1)
         (null angle-of-new-line-with-normal-2))
   (princ "solution to outbound line is complex - aborted")(terpri)
   (return-from find-outbound-line-2 '"complex")))

(cond ((<= (abs (- angle-of-new-line-with-normal-1 theta-out))
         (abs (- angle-of-new-line-with-normal-2 theta-out)))
   (return-from find-outbound-line-2
     (make-line
       point-intersect
       (find-point-from-coef-and-point
       point-intersect
       (car two-vectors)))))
      (t (return-from find-outbound-line-2
       (make-line
             point-intersect
        (find-point-from-coef-and-point
          point-intersect
          (cadr two-vectors)))))))))



(defun find-point-from-coef-and-point (point coef)    ;finds a point on a line
                                            ;with coef i, j, k and point.
  (let* ((end-point-line
      (init-point (map 'list '+ (send (eval Point) :list-format)
                            (map 'list '* '(100 100 100)
                              coef)))))
    end-point-line))
```

131

```lisp
(defun make-unit-line (point line)   ;makes a unit line from a point
                                     ; parallel to line
   (let* ((unit-vector (send (eval (line-segment-direction-vector (eval line)))
                       :unit-vector))
       (point-coord (send (eval point) :list-format)))

      (make-line point
            (init-point
             (list (+ (first unit-vector)(first point-coord))
                   (+ (second unit-vector)(second point-coord))
                   (+ (third unit-vector)(third point-coord)))))))))

(defun parallel-lines (line-1 line-2)   ; returns 't if parallel, nil if not
   (let ((Ti1 (vector-i (eval (line-segment-direction-vector (eval line-1)))))
       (Ti2 (vector-i (eval (line-segment-direction-vector (eval line-2)))))
       (Tj1 (vector-j (eval (line-segment-direction-vector (eval line-1)))))
       (Tj2 (vector-j (eval (line-segment-direction-vector (eval line-2)))))
       (Tk1 (vector-k (eval (line-segment-direction-vector (eval line-1)))))
       (Tk2 (vector-k (eval (line-segment-direction-vector (eval line-2)))))
       (Tval 'nil))
      (cond ((and (not (zerop Ti1)) (not (zerop Ti2)))
          (setf Tval (/ Ti1 Ti2)))
         ((and (not (zerop Tj1)) (not (zerop Tj2)))
          (setf Tval (/ Tj1 Tj2)))
         ((and (not (zerop Tk1)) (not (zerop Tk2)))
          (setf Tval (/ Tk1 Tk2)))
         (t (return-from parallel-lines 'nil)))
      (cond ((and (equal Ti1 (* Tval Ti2))
             (equal Tj1 (* Tval Tj2))
             (equal Tk1 (* Tval Tk2)))
          (return-from parallel-lines 't))
         (t 'nil))))

(defun find-intercept-point-extended (plane line) ; find intercept point of a
                                                  ; plane and line segment
                                                  ; extended to reach the plane,
                                                  ; if it exists.
                                                  ; return NIL if not exist
   (let ((denom (rationalize (denom-in-intercept plane line)))
       (t-intercept 'nil)
       (I-point 'nil))
      (cond ((not (equal-zero-p denom))
          (setf t-intercept (rationalize (solve-for-t
                                   (send (eval plane) :list-coeff)
                                   line
                                   denom)))
          (setf I-point (get-intercept-point-2-extended line t-intercept))))
      I-point))

(defun get-intercept-point-2-extended (line t-intercept)
                                           ; return the name of a valid intercept
                                           ; point without checking that intercept
                                           ; point is on the actual line segment
   (let ((I 'nil)
       (I-list 'nil))
      (cond (t (setf I-list (vector-add-with-t
                       (line-segment-direction-vector (eval line))
                       (line-segment-position-vector (eval line))
                       t-intercept))
          (setf I (init-point I-list))))
      I))
```

132

```lisp
(defun normalize-vector (vector)
                          ;takes a vector i j k ... and normalizes these three
                          ; by dividing each by sqrt(ii + jj + kk)
   (let* ((i (first vector))
      (j (second vector))
      (k (third vector))
      (X (cdddr vector))
      (denomonator (sqrt (+ (* i i)(* j j)(* k k)))))
      (cons (/ i denomonator)(cons (/ j denomonator)(cons (/ k denomonator)
X))))))


(defun get-adjustment-vectors (point facet)   ;returns unit vectors
                                              ; 1 - point to center of facet and
                                              ; 2 - 90 deg off and in facet
   (let* ((line-N (make-unit-line point
                          (normal-line point facet)))
      (line-p (make-unit-line point
                          (make-line point
                                     (facet-center
                                        (eval facet)))))
      (equation-1 (append
                     (send
                      (eval (line-segment-direction-vector (eval line-N)))
                      :list-format)
                     '(0)))
      (equation-2 (append
                     (send
                      (eval (line-segment-direction-vector (eval line-p)))
                      :list-format)
                     '(0)))
      (Two-equations (list equation-1 equation-2))
      (two-vectors))
      (setf two-vectors(find-outbound-vector Two-equations))
      (cond ((null (caar two-vectors))
         (princ "Adjustment vectors returns complex roots")(terpri)
         (return-from get-adjustment-vectors "complex"))
         (t (list (send
                    (eval (line-segment-direction-vector (eval line-N)))
                    :list-format)
                   (car two-vectors)))))))


(defun check-facet-list-against-snells-law (line facet-list volume-list)

   (do*((start-point
      (vector-start-point (eval (line-segment-direction-vector (eval line))))
      (vector-start-point (eval (line-segment-direction-vector (eval line)))))
         (point (cond((null (find-intercept-point-extended (car facet-list) line))
               (return-from check-facet-list-against-snells-law 'nil))
             (t (find-intercept-point-extended (car facet-list) line)))
          (cond((null (find-intercept-point-extended (car facet-list) line))
               (return-from check-facet-list-against-snells-law 'nil))
             (t (find-intercept-point-extended (car facet-list) line))))
        (test-1 (cond((point-in-volume-P point (car volume-list)) 't)
             (t (return-from check-facet-list-against-snells-law 'nil)))
          (cond((point-in-volume-P point (car volume-list)) 't)
             (t (return-from check-facet-list-against-snells-law 'nil))))
```

133

```
          (line (find-outbound-line-2 (make-line start-point point)
                                 (car facet-list)
                                 (volume-probability-of-detection
                                  (eval (car volume-list)))
                                 (volume-probability-of-detection
                                  (eval (cadr volume-list))))
               (find-outbound-line-2 (make-line start-point point)
                                 (car facet-list)
                                 (volume-probability-of-detection
                                  (eval (car volume-list)))
                                 (volume-probability-of-detection
                                  (eval (cadr volume-list)))))
        (volume-list (cdr volume-list)(cdr volume-list))
        (facet-list (cdr facet-list)(cdr facet-list)))
       ((and (null facet-list)
        (not (or (null line)
                 (equal "complex" line)
                 (equal "reflect" line)
                 (equal "div-by-zero" line)))) line)
 (cond((or (null line)
          (equal "complex" line)
          (equal "reflect" line)
          (equal "div-by-zero" line))
   (return-from check-facet-list-against-snells-law 'nil)))))
```

134

```
;;; -*- Mode:Common-Lisp; Base:10 -*-
;**********************************************************************
;
; PATH PLANNING                  D.H. Lewis                25 Aug 88
; ------------------------------------------------------------------
; Modified                       L.R. WRENN                6 Mar 89
;
;  Contains the flavors, methods, and functions nessesary to conduct path
; planning.  Divided into three main sections; Flavors, A-star
; path planning, and path optimization.
;  The flavors section provides the essential path and agenda item flavors,
; and their associated method and support functions.
;  The A* search section conducts an a* search of the volumes, minimizing
; cost and visibility, and creates an initial path.
;  Finally, the optimization code optimizes the initial A* path according
; to snells law criteria.  This section may create one or several paths
;
;**********************************************************************
;
;     MAIN FUNCTIONS:
;           A-STAR-SEARCH
;           A-STAR-SEARCH-M
;
;     OTHER FUNCTIONS:
;           MAKE-PATH-NAME
;           INIT-NEW-PATH
;           MAKE-AGENDA-ITEM-NAME
;           INIT-AGENDA-ITEM
;           PUT-SUCCESSORS-ON-AGENDA
;           AGENDA-SORT-P
;           GOAL-ON-AGENDA-P
;           REMOVE-GOAL
;           FIND-PATH
;           PRINT-AGENDA
;           EVALUATION-FUNCTION
;           COST-FUNCTION
;           EVAL-TURN-COST
;           PROJECT-XY
;           FIND-PREVIOUS-VOLUME
;           EVAL-CLIMB-DIVE
;           CALC-PATH-AND-STATS
;           FIND-INTERMEDIATE-FACETS
;           MAKE-FACET-TO-FACET-PATH
;
;     OPTIMIZE PATH FUNCTIONS:
;           OPTIMIZE-PATH
;           OPTIMIZE-POINT-ON-FACET
;           OPTIMIZE-K-ON-LINE
;           FIND-EDGE-POINTS-OF-FACET
;           AGENDA-SORT-ON-K
;           FIND-SNELLS-CONSTANT
;
;**********************************************************************

(defvar *PD-threshold* '0.0)          ; maximum desirable probability of
detection
(defvar *PD-modifier* '10.0)          ; affects effect of PD on path planning
(defvar *PI* '3.14159)

(defvar *path-counter* '0)            ; path name variables
(defvar *list-of-paths* 'nil)         ; location of all instanciated paths
(defvar *agenda-counter* '0)          ; agenda instanciations
```

135

```
(defvar *Turn45* '10.0)              ; cost for turn of 45 degrees or less
(defvar *Turn90* '50.0)              ; cost for turn between 45 and 90 degrees
(defvar *BigTurn* '5000.0)           ; cost for turns greater than 90 degrees

(defvar *Shallow-Climb* '1.20)       ; ratio modifier for a shallow climb
(defvar *Steep-Climb* '1.80)         ; ratio modifier for a steep climb
(defvar *Dive* '0.80)                ; ratio modifier for any dive
(defvar Pt1)                         ; used by :make-polygon-list
(defvar Pt2)                         ; used by :make-polygon-list

(defvar *Start-fuel* '1225)          ; Fuel aircraft will start with
(defvar *Start-TAS* '450)            ; True Air Speed that the missile will
start with
(defvar *Fuel*)                      ; globle used to pass fuel remaining
between functions
(defvar *TAS*)                       ; globle used to pass current TAS between
functions


;*****************************************************************************
;
;        F L A V O R S ,   M E T H O D S ,   A N D    F U N C T I O N S
;
;*****************************************************************************


;----------------------------------------------------
;                   PATH FLAVOR
;----------------------------------------------------

(defflavor path
        (start-point
      end-point                    ; goal
      volumes                      ; general path "corridor"
      facets                       ; "windows" in cooridor
      lines                        ; specific path to follow
      points                       ; turn points in path
      length                       ; of current lines
      total-K                      ; sum of deviations from snells law for path
      max-detection-probability
      ave-detection-probability)   ; average of entire path corridor
     (graphic)
   :gettable-instance-variables
   :settable-instance-variables
   :inittable-instance-variables
   :outside-accessible-instance-variables)

;--------------METHODS FOR PATHS----------------------

(defmethod (path :length) ()       ; find the total length of the path
  (let ((val '0.0))
     (cond ((null length)
            (loop for L in lines
              do (setf val (+ val (send (eval L) :length))))
            (setf length val)))
     length))

(defmethod (path :max-detection-probability) ()  ; find the highest PD on the
path
  (let ((maximum (volume-probability-of-detection (eval (first volumes)))))
     (loop for V in (rest volumes)
       do (cond ((< maximum (volume-probability-of-detection (eval V)))
            (setf maximum (volume-probability-of-detection (eval V))))))
     (setf max-detection-probability maximum)))
```

```lisp
(defmethod (path :ave-detection-probability) ()   ; find the weighted average of
the PD's
   (let ((weighted-sum '0.0))
     (loop for Counter from 0 to (1- (length volumes))
       do (setf weighted-sum
             (+ weighted-sum
               (* (send (eval (nth Counter lines)) :length)
                   (volume-probability-of-detection (eval (nth Counter
Volumes)))))))))
     (setf ave-detection-probability (/ weighted-sum
                                (send self :length)))
   ave-detection-probability))

(defmethod (path : make-node-list) ()        ; used by graphic mixin-flavor to
draw
   (loop for P in points
     collect (reverse (append (list '1) (reverse (send (eval P)
:list-format))))))

(defmethod (path :make-polygon-list) ()      ; used by graphic mixin-flavor to
draw
   (loop for L in lines
     do (setf Pt1 (car (send (eval L) :endpoint-list)))
     do (setf Pt2 (cadr (send (eval L) :endpoint-list)))
       collect (list (position-if '(lambda (A) (equal A Pt1)) node-list)
                 (position-if '(lambda (A) (equal A Pt2)) node-list))))

;---------------------PATH NAMES---------------------


(defun make-path-name ()           ; make a new name for a path
   (gensym (incf *path-counter*))
   (intern (gensym "path")))

(defun init-new-path (start end volumes facets lines points length K) ;make a
new path
   (let ((name (make-path-name)))
     (set name (make-instance 'path
                   :start-point start
                   :end-point end
                   :volumes volumes
                   :facets facets
                   :lines lines
                   :points points
                   :length length
                   :total-K K
                   :max-detection-probability 'nil
                   :ave-detection-probability 'nil))
     (push name *list-of-paths*)
     name))
```

137

```
;-------------------------------------------------------
;                    AGENDA-ITEM FLAVOR
;-------------------------------------------------------

(defflavor agenda-item
    (volume
     cost
     evaluation
     path
     fuel
     TAS)
    ()
  :gettable-instance-variables
  :settable-instance-variables
  :inittable-instance-variables
  :outside-accessible-instance-variables)




;------------------AGENDA-ITEM NAMES------------------

(defun make-agenda-item-name ()
  (gensym (incf *agenda-counter*))
  (intern (gensym "agenda")))

(defun init-agenda-item (volume cost evaluation path fuel TAS)
  (let ((name (make-agenda-item-name)))
    (set name (make-instance 'agenda-item
                   :volume volume
                   :cost cost
                   :evaluation evaluation
                   :path path
                   :fuel fuel
                   :TAS TAS))
    name))


;*********************************************************************
;
;        S     E     A     R     C     H     E     S
;
;*********************************************************************

;-------------------------------------------------------
;                    A* Search
;-------------------------------------------------------


(defun A-star-search (Start-point End-point Trace-flag Camera-flag)
  (let* ((start-volume (first (locate-point-air start-point)))
     (goal-volume (first (locate-point-air end-point)))
     (successor-volumes (volume-connected-to (eval start-volume)))
     (path-volumes 'nil)
     (agenda 'nil)
     (best-path)
     (ground-volumes 'nil))

    (terpri) (terpri)
    (princ ">>>>Begin A-star Search") (terpri) (terpri)
    (princ "    Start Volume: ") (prin1 start-volume) (terpri)
    (princ "    Goal Volume: ") (prin1 goal-volume) (terpri) (terpri)
```

138

```lisp
(cond (trace-flag
   (terpri) (princ "Search trace selected.  Top five and bottom five items")
   (terpri) (princ "on seach agenda will be printed.") (terpri) (terpri))
 (t (terpri)))
(cond (Camera-flag
   (terpri) (princ "Display the search as it progresses has been selected")
   (terpri) (princ "reduce and move the lisp listener window to")
   (terpri) (princ "the right 1/4 of the screen. press <RETURN> when done")
   (wait-for-keyboard-input) (terpri) (terpri)
   (movie-ground)
   (loop for V in (universe-volumes *universe*)
        do (cond ((equal 'ground (volume-composition (eval V)))
                (setf ground-volumes (adjoin V ground-volumes)))))))
 (t (terpri)))
(princ "Search")

                         ; initalize the search agenda
(setf *fuel* *Start-fuel*)                          ; init *fuel* for new path
(setf *TAS* *Start-TAS*)                            ; init *TAS* for new path
(setf agenda (put-successors-on-agenda
                        start-volume          ; end of last path
                   successor-volumes   ; successors to be added
                   (init-cost start-volume
                              start-point
                              trace-flag)    ; cost
                   (list start-volume) ; path to date
                   end-point               ; goal
                   agenda))                ; agenda to be changed

                   ; SEARCH along best agenda item for all possible paths
                   ; until get to the goal along one of the paths

(loop until (goal-on-agenda-p goal-volume agenda)
  do (princ ".")
  do (cond (trace-flag
        (princ "--------------New Agenda--------------------")
             (print-agenda agenda)))
  do (cond (camera-flag
        (display-movie-path agenda start-point ground-volumes)))

  do (let* ((best-successor-volume (first agenda))
        (successors-to-best (volume-connected-to (eval (agenda-item-volume
                                          (eval best-successor-volume)))))))
     (setf successors-to-best (remove 'EDGE successors-to-best))
     (loop for V in (rest (agenda-item-path (eval best-successor-volume)))
       do (setf successors-to-best (remove V successors-to-best)))
     (setf agenda (remove best-successor-volume agenda))
                             ;set *fuel* and *tas* from
best-successor-volume
     (setf *fuel* (agenda-item-fuel (eval best-successor-volume)))
     (setf *TAS* (agenda-item-tas (eval best-successor-volume)))
     (setf agenda (put-successors-on-agenda
                   (agenda-item-volume (eval best-successor-volume))
                   successors-to-best
                   (agenda-item-cost (eval best-successor-volume))
                   (agenda-item-path (eval best-successor-volume))
                   end-point
                   agenda))))

                   ; SEARCH COMPLETED!
```

139

```
                         ; find lines and points in search

(cond (camera-flag
   (display-movie-path agenda start-point ground-volumes)))
(setf path-volumes (reverse (find-path goal-volume agenda)))
                                           ;get resultant path
(setf best-path (init-new-path start-point
                      end-point
                      path-volumes
                      'nil
                      'nil
                      'nil
                      'nil
                      'nil))
(princ "Completed") (terpri) (terpri)


(make-facet-to-facet-path best-path)    ; make initial guess at optimal path
(calc-path-and-stats best-path)         ; fill out rest of path flavor data
(cond (camera-flag
   (send (eval best-path) :initialize)
   (loop for N in '(1 2 3 4)
      do (cond ((not (equal N 4))
              (send (eval (nth N *list-of-VCRs*)) :clear-scene)))
      do (show-path-4 (nth N *list-of-VCRs*)
                  (nth N *window-stats*)
                  best-path
                  (first ground-volumes)
                  (nth N *display-stats*))
      do (cond ((not (equal N 4))
              (send (eval (nth N *list-of-VCRs*))
                  :display-label best-path))))))

best-path))
```

```
;----------------------------------------------------------
;       A* Search with multiple solutions
;----------------------------------------------------------


(defun A-star-search-M (Start-point End-point Trace-flag paths Camera-flag)
  (let* ((start-volume (first (locate-point-air start-point)))
     (goal-volume (first (locate-point-air end-point)))
     (successor-volumes (volume-connected-to (eval start-volume)))
     (path-volumes 'nil)
     (agenda 'nil)
     (paths-found)
     (ground-volumes 'nil))

    (terpri)
    (princ ">>>>Begin A-star Search") (terpri) (terpri)
    (princ "      Start Volume: ") (prin1 start-volume) (terpri)
    (princ "       Goal Volume: ") (prin1 goal-volume) (terpri) (terpri)
    (cond (trace-flag
      (terpri)
      (princ "Search trace selected.  Top five and bottom five items")
      (terpri) (princ "on seach agenda will be printed.") (terpri) (terpri))
     (t (terpri)))
    (cond (Camera-flag
      (terpri)
      (princ "Display the search as it progresses has been selected")
      (terpri) (princ "reduce and move the lisp listener window to")
      (terpri)
      (princ "the right 1/4 of the screen. press <RETURN> when done")
      (wait-for-keyboard-input) (terpri) (terpri)
      (movie-ground)
      (loop for V in (universe-volumes *universe*)
          do (cond ((equal 'ground (volume-composition (eval V)))
                  (setf ground-volumes (adjoin V ground-volumes)))))))
     (t (terpri)))

                      ; initalize the search agenda

    (setf *fuel* *Start-fuel*)              ; init *fuel* for new path
    (setf *TAS* *Start-TAS*)                ; init *TAS* for new path
    (setf agenda (put-successors-on-agenda
                      start-volume           ; end of last path
                    successor-volumes   ; successors to be added
                    (init-cost start-volume
                          start-point
                          trace-flag)    ; cost
                    (list start-volume) ; path to date
                    end-point           ; goal
                    agenda))            ; agenda to be changed

                 ; SEARCH along best agenda item for all possible paths
                 ; until get to the goal along one of the paths

    (loop repeat paths      ; find top several paths
      do (terpri)
      do (princ "Search")
      do (loop until (goal-on-agenda-p goal-volume agenda)
                                      ; same loop as single search
          do (princ ".")
          do (cond (trace-flag
              (princ "--------------New Agenda--------------------")
                  (print-agenda agenda)))
```

```
do (cond (camera-flag
          (display-movie-path agenda start-point ground-volumes)))
   do (let* ((best-successor-volume (first agenda))
             (successors-to-best (volume-connected-to
                                  (eval (agenda-item-volume
                                         (eval best-successor-volume)))))))
          (setf successors-to-best (remove 'EDGE successors-to-best))
          (loop for V in (rest
                          (agenda-item-path
                           (best-successor-volume)))
            do (setf successors-to-best (remove V successors-to-best)))
          (setf agenda (remove best-successor-volume agenda))
          (setf agenda (put-successors-on-agenda
                        (agenda-item-volume
                         (eval best-successor-volume))
                        successors-to-best
                        (agenda-item-cost
                         (eval best-successor-volume))
                        (agenda-item-path
                         (eval best-successor-volume))
                        end-point
                        agenda))))
do (cond (camera-flag
          (display-movie-path agenda start-point ground-volumes)))

(setf path-volumes (reverse (find-path goal-volume agenda)))
(setf agenda (remove-goal goal-volume agenda))
(setf paths-found (adjoin (init-new-path start-point
                                         end-point
                                         path-volumes
                                         'nil
                                         'nil
                                         'nil
                                         'nil
                                         'nil)
                   paths-found))

(princ "Completed") (terpri) (terpri)
(make-facet-to-facet-path (first paths-found))
(calc-path-and-stats (first paths-found))
(cond (camera-flag
  (send (eval (first paths-found)) :initialize)
  (loop for N in '(1 2 3 4)
        do (cond ((not (equal N 4))
                  (send (eval (nth N *list-of-VCRs*)) :clear-scene)))
        do (show-path-4 (nth N *list-of-VCRs*)
                        (nth N *window-stats*)
                        (first paths-found)
                        (first ground-volumes)
                        (nth N *display-stats*))
        do (cond ((not (equal N 4))
                  (send (eval (nth N *list-of-VCRs*))
                        :display-label (first paths-found)))))))))
```

142

```lisp
      (cond (camera-flag
          (loop for N in '(1 2 3 4)
                do (cond ((not (equal N 4))
                     (send (eval (nth N *list-of-VCRs*)) :clear-scene))))
          (loop for P in paths-found
                do (send (eval P) :initialize)
                do (loop for N in '(1 2 3 4)
                     do (show-path-4 (nth N *list-of-VCRs*)
                                     (nth N *window-stats*)
                                     P
                                     (first ground-volumes)
                                     (nth N *display-stats*))
                        do (cond ((not (equal N 4))
                             (send (eval (nth N *list-of-VCRs*))
                               :display-label (first paths-found))))))))))


    paths-found))


;----------------------------------------------------
;            Search utility functions
;----------------------------------------------------


;-------agenda manipulations-------

                                    ; for A-STAR search
(defun put-successors-on-agenda (pred-volume
                           successor-volumes
                           cost                    ; cost so far
                           path                    ; volumes
                           goal
                           agenda)
  (loop for V in successor-volumes
    do (setf agenda (adjoin
               (init-agenda-item V                  ; name
                            (+ cost (cost-function V path))
                            (evaluation-function pred-volume
                                            V
                                            path
                                            goal)
                            (adjoin V path)    ; path
                            *fuel*    ; altered during cost-function
                            *TAS*)    ; altered during cost-function
                      agenda)))
  (stable-sort agenda #'agenda-sort-p))

(defun agenda-sort-p (A B)
  (cond ((LT (+ (agenda-item-cost (eval A))
                (agenda-item-evaluation (eval A)))
             (+ (agenda-item-cost (eval B))
                (agenda-item-evaluation (eval B))))
         (return-from agenda-sort-p 't)))
  'nil)

(defun goal-on-agenda-p (goal agenda)    ; return T iff goal volume is on the
agenda
  (loop for A in agenda
    do (cond ((equal goal (agenda-item-volume (eval A)))
         (return-from goal-on-agenda-p 'T))))
  'nil)
```

```lisp
(defun remove-goal (goal agenda)
  (loop for A in agenda
    do (cond ((equal goal (agenda-item-volume (eval A)))
         (return-from remove-goal (remove A agenda)))))
  'nil)

(defun find-path (goal agenda)          ; get the path once the goal is found
  (loop for A in agenda
    do (cond ((equal goal (agenda-item-volume (eval A)))
         (return-from find-path (agenda-item-path (eval A)))))))

(defun print-agenda (agenda)            ; print agenda and some/all items on the
agenda
  (terpri) (pprint agenda) (terpri)
  (cond ((>= 10 (length agenda))
     (princ "Entire agenda: ") (terpri)    ; print whole agenda if short
       (loop for I in agenda
             do (terpri)
             do (describe I)))
    (t (princ "First five in agenda: ") (terpri) ; do first five and last five
       (loop for Count in '(0 1 2 3 4)            ; if long
         do (describe (nth count agenda))
         do (terpri))
       (terpri) (princ "Last five on agenda: ") (terpri)
       (loop for Count in '(6 5 4 3 2 1)
         do (describe (nth (- (length agenda) Count) agenda))
         do (terpri))))
    (terpri) (terpri))


;-----evaluation and cost functions------

(defun evaluation-function (VP VS path-volumes Goal)
  (let (;(turn-modifier (eval-turn-cost VP VS path-volumes))
    ;(altitude-modifier (eval-climb-dive VP VS))
    (PD-modifier  '1.0 )  ;(* *PD-modifier*
                     ;(- (volume-probability-of-detection (eval VS))
                     ;   *PD-threshold*))))
    (basis-distance (distance (volume-arithmetic-center (eval VS)) Goal)))
    (setf PD-modifier  basis-distance)))


(defun init-cost (VStart start-point trace-flag)
  (let ((PD-modifier (* 100
                 (volume-probability-of-detection (eval VStart))
                 (/ (distance start-point
                       (volume-arithmetic-center (eval VStart)))
                   (/ 450 60)))))
    (basis-cost (fuel-burned (distance
                       (volume-arithmetic-center (eval VStart))
                          start-point)
                     (climb-angle start-point
                             (volume-arithmetic-center (eval VStart)))
                     *fuel*
                     *TAS*)))
    (cond (trace-flag
      (terpri)(princ "The initial cost of the search from the")
      (terpri)(princ "start point to the volume it is in center is")
      (terpri)(princ (+ PD-modifier basis-cost))))
    (+ PD-modifier basis-cost)))
```

144

```lisp
(defun cost-function (VS path-volumes)
  (let (        ;(altitude-modifier (eval-climb-dive (first path-volumes) VS))
      (turn-modifier (eval-turn-cost (first path-volumes) VS path-volumes))
      (PD-modifier (PD-cost (first path-volumes)
                        (volume-arithmetic-center (eval (first path-volumes)))
                        VS
                        (volume-arithmetic-center (eval VS))))
     (basis-cost (fuel-burned (distance (volume-arithmetic-center (eval VS))
                              (volume-arithmetic-center (eval (first
path-volumes))))
                        (climb-angle (volume-arithmetic-center (eval (first
path-volumes)))
                              (volume-arithmetic-center (eval VS)))
                   *fuel*
                   *TAS*)))
    (+ turn-modifier PD-modifier basis-cost)))

(defun PD-cost (VP VP-point VS VS-point)
  (let* ((common-facet (find-common-facet VP VS))
      (straight-line (make-line VP-point VS-point))
      (intercept-point (find-intercept-point common-facet straight-line)))
    (+ (* 100
      (volume-probability-of-detection (eval VP))
      (/ (distance VP-point intercept-point)
         (/ 450 60)))
       (* 100
      (volume-probability-of-detection (eval VS))
      (/ (distance VS-point intercept-point)
         (/ 450 60)))))))

(defun eval-turn-cost (VP VS Path-volumes)
  (let ((projected-VP-center (project-xy (volume-arithmetic-center (eval VP))))
      (projected-VS-center (project-xy (volume-arithmetic-center (eval VS))))
      (previous-volume (find-previous-volume VP Path-volumes))
      (projected-vol-center 'nil)
      (path 'nil)
      (new-path 'nil)
      (angle-of-turn 'nil))

    (cond ((equal VP previous-volume)                ; no previous path ?
        (return-from eval-turn-cost '1.0))
       (t (setf projected-vol-center (project-xy
                        (volume-arithmetic-center (eval previous-volume))))
          (setf path (make-line projected-vol-center projected-VP-center))
          (setf new-path (make-line projected-VP-center projected-VS-center))
          (setf angle-of-turn (angle-between-lines path new-path))
          (cond ((GT angle-of-turn (/ *PI* '2.0))
              (return-from eval-turn-cost (* (- angle-of-turn 90) 2)))))))  ;
turn > 90
     *Turn45*))                                      ; turn <= 90


(defun project-xy (Point)
  (let ((point-coords (send (eval Point) :list-format)))
    (init-point (list (first point-coords) (second point-coords) '0.0)))))

(defun find-previous-volume (VP path-volume)
  (let ((position-VP (position VP path-volume)))
    (cond ((> 1 (length path-volume))
        (return-from find-previous-volume (elt (1+ position-VP)
path-volume)))
       (t (return-from find-previous-volume (first path-volume))))))
```

145

```lisp
(defun eval-climb-dive (VP VS)
  (let* ((inter-facet (find-common-facet VP VS))
         (interfacet-z (third (mean-point-in-facet inter-facet)))
         (path-z (third
                   (send (eval (volume-arithmetic-center (eval VP))) :list-format)))))
    (cond ((and (LT path-z (* interfacet-z '1.10))
                (GT path-z (* interfacet-z '0.90)))
           (return-from eval-climb-dive '1.0))          ; level flight
          ((GT path-z interfacet-z)
           (return-from eval-climb-dive *Dive*))          ; dive
          (t (loop for P in (send (eval inter-facet) :points)
                do (cond ((> path-z (third (send (eval P) :list-format)))
                                                          ; shallow climb
                          (return-from eval-climb-dive *Shallow-Climb*))))))
    *Steep-Climb*))                                       ; steep climb


;------general functions in support of path planning------


(defun Calc-path-and-stats (path)          ; used to find support info on a new
path
  (send (eval path) :length)

                                           ; determine probability limits

  (send (eval path) :max-detection-probability)
  (send (eval path) :ave-detection-probability)
  (princ ">>>>Path Statistics:") (terpri) (terpri)
  (princ "   Maximum detection probability: ")
  (prin1 (path-max-detection-probability (eval path)))
  (terpri)
  (princ "   Average detection probability: ")
  (prin1 (path-ave-detection-probability (eval path)))
  (terpri)
  (princ "   Total length of path: ")
  (prin1 (path-length (eval path)))
  (terpri)
  (princ "   Total number of maneuvers: ") (prin1
                                  (- (length (path-points (eval path))) '2))
  (terpri) (terpri)
  (princ ">>>>Path: ") (prin1 path) (terpri) (terpri)

  'nil)

(defun find-intermediate-facets (path)              ; find all the facets along
                                                    ; the path
  (let ((previous-volume (first (path-volumes (eval path))))
        (facets 'nil))
    (loop for V in (rest (path-volumes (eval path)))
      do (setf facets (adjoin (find-common-facet previous-volume V) facets))
      do (setf previous-volume V))
    (reverse facets)))

(defun make-center-to-center-path (path)
  (let ((last-point (path-start-point (eval path)))
        (points (path-start-point (eval path)))
        (lines 'nil))
    (setf (path-facets (eval path)) (find-intermediate-facets path))
    (loop for V in (path-volumes (eval path))
```

146

```lisp
       do (let ((next-point (volume-arithmetic-center (eval V))))
             (setf lines (adjoin (make-line last-point next-point) lines))
             (setf points (adjoin next-point points))
             (setf last-point next-point)))
      (push (make-line last-point (path-end-point (eval path))) lines)
      (push (path-end-point (eval path)) points)
      (setf (path-lines (eval path)) (reverse lines))
      (setf (path-points (eval path))
        (adjoin (path-start-point (eval path)) (reverse points)))))


(defun make-facet-to-facet-path (path)
  (let ((last-point (path-start-point (eval path)))
     (points (path-start-point (eval path)))
     (lines 'nil))
     (setf (path-facets (eval path)) (find-intermediate-facets path))
     (loop for F in (path-facets (eval path))
       do (let ((next-point (init-point (mean-point-in-facet F))))
         (setf lines (adjoin (make-line last-point next-point) lines))
         (setf points (adjoin next-point points))
         (setf last-point next-point)))
      (push (make-line last-point (path-end-point (eval path))) lines)
      (push (path-end-point (eval path)) points)
      (setf (path-lines (eval path)) (reverse lines))
      (setf (path-points (eval path))
        (adjoin (path-start-point (eval path)) (reverse points)))))


(defun get-keyboard-input ()
  (send *terminal-io* :tyi-no-hang))


(defun wait-for-keyboard-input ()
  (send *terminal-io* :tyi))
```

147

```
;**********************************************************************
;
;                    P A T H   O P T I M I Z A T I O N
;
;**********************************************************************
;;;
;;;   OPTIMIZE PATH ACCORDING TO SNELL'S LAW.            D.H. LEWIS  10/11/88
;;;
;;;     Develop an expression for snell's constant at each facet along the
;;;   the path, and then minimize it with respect to the facets before
;;;   and after the facet concerned.  Sum all constants along the path
;;;   to determine the net amount of deviation from snell's law. Repeat
;;;   until total constant minimized.
;;;
;**********************************************************************
*

(defvar *PI2* (/ *PI* '2.0))
(defvar *search-increment* '10)

;-------------------MAIN PATH OPTIMIZATION FUNCTION-------------------

(defun optimize-path (path)
  (let ((new-path-points (list (path-start-point (eval path))))
    (new-path-lines 'nil)
    (new-path-length '0.0)
    (new-path 'nil)
    (last-point 'nil)          ; dummy for building path lines
    (total-K '0.0))            ; total deviation from snell's law

                              ; optimize path point for each facet in turn,
                              ; appending new points onto new-point list as
                              ; they are created

    (terpri) (terpri)
    (princ "Optimizing path ") (prin1 path) (princ ":") (terpri) (terpri)
    (loop for Facet-nr from '1 to (length (path-facets (eval path)))
      do (let ()
      (princ "Optimizing at facet number ")
            (prin1 facet-nr) (princ " : ") (prin1 (nth (1- facet-nr) (path-facets
(eval path)))))
      (terpri))
      do (let ((prev-point 'nil)
            (next-point (nth (1+ facet-nr) (path-points (eval path))))
            (path-point (nth facet-nr (path-points (eval path))))
            (new-point 'nil)
            (facet (nth (1- facet-nr) (path-facets (eval path))))
            (N1 (+ '1 (volume-probability-of-detection
                        (eval (nth (1-  facet-nr) (path-volumes (eval path)))))))
            (N2 (+ '1 (volume-probability-of-detection
                        (eval (nth facet-nr (path-volumes (eval path))))))))

                        ; use "best" previous point estimate

      (cond ((> facet-nr '1)
            (setf prev-point (first new-path-points)))
            (t (setf prev-point (nth (1- facet-nr) (path-points (eval path)))))))

;      (pprint (list '"initial: " facet-nr prev-point path-point next-point
facet N1 N2))
```

148

```lisp
        (setf new-point (optimize-point-on-facet prev-point
                                        next-point
                                        facet
                                        path-point
                                        N1
                                        N2))
;          (pprint (list '"new path point: " new-point (get new-point 'K)))
          (setf new-path-points (adjoin new-point new-path-points))
          (setf total-K (+ total-K (get new-point 'K)))))

                                        ; add goal to new points, draw new path

      (setf new-path-points (adjoin (car (last (path-points (eval Path))))
  new-path-points))
      (setf new-path-points (reverse new-path-points))
      (setf last-point (first new-path-points))
      (loop for P in (rest new-path-points)
        do (let ()
        (setf new-path-lines (adjoin (make-line last-point P) new-path-lines))
        (setf new-path-length (+ (send (eval (first new-path-lines)) :length)
                        new-path-length))
        (setf last-point P)))
      (setf new-path-lines (reverse new-path-lines))

                                        ; build the new path with optimized path data

      (terpri) (terpri)
      (princ "Optimization completed") (terpri)
      (setf new-path(init-new-path (path-start-point (eval path))
                                (path-end-point (eval path))
                                (path-volumes (eval path))
                                (path-facets (eval path))
                                new-path-lines
                                new-path-points
                                new-path-length
                                total-K))
      (calc-path-and-stats new-path)
      new-path))


;----------------FIND THE BEST POINT ON THE FACET------------------

(defun optimize-point-on-facet (prev-point next-point facet path-point N1 N2)

                                        ; Find the point on the facet with the lowest
                                        ; snell's constant (K).

  (let* ((straight-path-line (make-line prev-point next-point))
        (straight-path-point (find-intercept-point facet straight-path-line))
        (path-K-line (make-line path-point straight-path-point))
        (path-plane (make-a-plane prev-point path-K-line))
        (list-of-points (find-edge-points-of-facet path-plane facet)))
;        (pprint list-of-points)
;        (pprint (list facet straight-path-point))
        (setf (get straight-path-point 'K) (find-snells-constant
                                        straight-path-point
                                        (make-line straight-path-point  prev-point)
                                        (make-line straight-path-point  next-point)
                                        facet
                                        N1
                                        N2))
```

149

```
                                      ; do special cases first

        (cond ((inside-facet-p straight-path-point facet)
            (cond ((equal '0.0 (* '1.0 (get straight-path-point 'K)))
                   (return-from optimize-point-on-facet straight-path-point))
                  (t (setf list-of-points (adjoin straight-path-point
list-of-points)))))
          (t (setf list-of-points (adjoin path-point list-of-points))))
;       (pprint (list list-of-points (length list-of-points)))
        (cond ((< '1 (length list-of-points))
              (setf path-point (optimize-K-on-line list-of-points
                                    prev-point
                                    next-point
                                    facet
                                    N1
                                    N2)))
           (t (setf (get path-point 'K) (find-snells-constant Path-point
                                      (make-line Path-point prev-point)
                                      (make-line Path-point next-point)
                                      facet
                                      N1
                                      N2))))
        path-point))


(defun optimize-K-on-line (agenda prev-point next-point facet N1 N2)
    (let ((lowest-K-point 'nil)
      (best-line 'nil)
      (mid-point 'nil))
;       (pprint (list '"Optimize: " agenda))
        (loop for P in agenda
          do (setf (get P 'K) (find-snells-constant P
                                      (make-line P prev-point)
                                      (make-line P next-point)
                                      facet
                                      N1
                                      N2)))
        (setf agenda (stable-sort agenda #'agenda-sort-on-K))
        (setf lowest-K-point (first agenda))
;       (pprint (list '"Sorted optimize: " agenda lowest-K-point))

        (loop repeat '3
          do (let ()
          (setf best-line (make-line (first agenda) (second agenda)))
          (setf mid-point (init-point (send (eval best-line) :midpoint)))
          (setf (get mid-point 'K) (find-snells-constant mid-point
                                      (make-line mid-point prev-point)
                                      (make-line mid-point next-point)
                                      facet
                                      N1
                                      N2))
          (setf agenda
                (stable-sort (list (first agenda) (second agenda) mid-point)
                             #'agenda-sort-on-K))
;         (pprint agenda)
;         (pprint (list (first agenda) (get (first agenda) 'K)))
          ))

        (first agenda)))
```

```lisp
(defun find-edge-points-of-facet (plane facet)
  (let ((intercept-points 'nil))
    (loop for E in (facet-edges (eval facet))
      do (let ((intercept-point (find-intercept-point plane E)))
        (cond ((not (null intercept-point))
              (setf intercept-points (adjoin intercept-point
intercept-points))))))
  intercept-points))


(defun agenda-sort-on-K (A B)    ; sort by increasing absolute value of K
property
  (< (abs (get A 'K)) (abs (get B 'K))))



;------------------FIND SNELLS CONSTANT------------------


(defun find-snells-constant (Point Line-1 Line-2 Facet N1 N2)
                            ; find snells constant at a boundary, i.e.:
                            ;
                            ;        K = N1 * sin(theta-1) - N2 * sin(theta-2)
                            ;
                            ; note: returns NIL if anything would "blow this up"

  (let* ((end-point-normal-line
      (init-point (map 'list '+ (send (eval Point) :list-format)
                          (map 'list '* '(100 100 100)
                          (send (eval facet) :list-coeff-3)))))
    (normal-line (make-line Point end-point-normal-line))
    (perpendicular-plane
      (make-a-plane
        (init-point (list '0 '0 (third (send (eval point) :list-format))))
        normal-line))
    (line-joining-points (make-line (send (eval line-1) :end-point)
                                    (send (eval line-2) :end-point)))
    (default '100)
    (theta-1 (angle-between-lines Line-1 normal-line))
    (theta-2 (angle-between-lines Line-2 normal-line)))
    (cond ((and (n  (null Theta-1))
                (not (nu.  theta-2)))
      (setf theta-1 (abs (realpart theta-1)))
      (setf theta-2 (abs (realpart theta-2)))
          (cond ((< *PI2* theta-1)
            (setf theta-1 (- *PI* theta-1))))
      (cond ((< *PI2* theta-2)
            (setf theta-2 (- *PI* theta-2))))
      (cond ((> theta-1 (realpart (asin (/ N2 N1))))         ; critical angle?
            (setf theta-2 *PI2*)))
      (cond ((send (eval line-joining-points) :strattle-plane-p
perpendicular-plane)
                (return-from
                    find-snells-constant (- (* N1 (sin theta-1))
                                          (* N2 (sin theta-2)))))
            (t (return-from
                    find-snells-constant (- (* N1 (sin theta-1))
                                          (* N2 (- (* '2 *PI*)
                                                  (sin theta-1))))))))))
    default))
```

```lisp
;;; -*- Mode: Lisp ; Syntax : Common-lisp -*-
;*****************************************************************************
;
;                    M A I N   C O N T R O L   F U N C T I O N S
;            ------------------------------------------------------------
;
;  These functions perform overall control of the static construction
; phase of the code.  They include the initial input control loops (for
; both input methods, and the control loop for the visibility region
; construction phase of the static construction.  The initial set-up
; functions are first, followed by the middle phase set-up functions, large
; scale control functions, and finally, the actual input methods themselves.
;
; THESIS                                              D. H. LEWIS  20 OCT 88
;*****************************************************************************
;
;        ROUTINE TO INPUT A DATA STREAM AND CONSTRUCT THE VOLUME(S)
;
;   THESIS/CS4452                                     D.H. LEWIS    15 MAY 88
;      --------------------------------------------------------------
;
;   Builds the standard static flavors (Universe, above, below, and cameras),
; opens and reads input file, and carries the static phase through making
; air volumes convex.
;
; MAIN FUNCTIONS:    SET-UP (METHOD FILE)
;                    SET-UP-2
;
; OTHER FUNCTIONS:   INITIALIZE-VOLUME
;                    MAKE-VOLUME-WITH-FACET-DATA
;                    DECREASSING-SORT-ON-X-P
;                    DECREASING-SORT-ON-Y-P
;                    PRINT-HEADER-1
;
;*****************************************************************************


(defvar *Universe*)             ; location of names for all flavors used in
static
                                ; construction
(defvar *above*)                ; standard volumes used by intercept functions
(defvar *below*)                ;
(defvar *input-stream*)         ; system name for non-standard input file
(defvar *output-stream*)        ; system name for non-standard output file
(defvar *max-altitude* '1000)   ; maximum altitude in Input Method 2
(defvar *min-altitude* '0)      ; minimum altitude for Input Method 2
(defvar *not-convex-volumes*)   ; flag variable for Input Method 2 which tells
                                ; which facet building function(s) to use

(defvar *original-input-volumes* 'nil)   ; save various "states"
(defvar *convex-volumes* 'nil)
(defvar *final-visibility-regions* 'nil)
```

```
;----------------INITIAL SETUP--------------

(defun set-up (Method File)
  (print-header-1)
  (make-origin)                                    ; make favorite constants
  (make-null-vector)
  (setf *above* (make-instance 'volume))
  (setf *below* (make-instance 'volume))
  (setf *not-convex-volumes* 't)
  (setf *done-making-new-visibility-volumes-flag* 'nil)
  (setf *precision* '0.0025)
  (setf *large-integer* '1e4)
  (setf *list-of-error-planes* 'nil)
  (princ ">Constants Initialized") (terpri)
  (make-cameras)                                   ; create camera
  (princ ">Camera built") (terpri)
  (make-movie-cameras)                             ; create movie camera
  (princ ">Movie Camera built") (terpri)
  (setf *Universe* (make-instance 'Universe        ; create universe for
volumes
                    :Volumes '()
                    :observers '()))
  (princ ">Universe created; reading data file") (terpri) (terpri)
  (setq *input-stream* (open File :direction :input))

                              ; select and use input method

  (cond ((equal '1 Method)          ; volume oriented input method
         (do ((data (read *input-stream* nil 'done) ; read all volumes into
universe
               (setf data (read *input-stream* nil 'done))))
            ((atom data))
          (push (init-volume data) (universe-volumes *Universe*))
          (princ ">>>> Volume created: ")
          (prin1 (car *list-of-volumes*))
          (princ "   Composition: ")
          (prin1 (volume-composition (eval (car *list-of-volumes*)))) (terpri)
          (make-all-facets (car *list-of-volumes*))) ; make all facets for new
volume
     (loop for V in (universe-volumes *universe*)
       do (setf (volume-visibility (eval V)) 'nil)))  ; remove ambiant
visibility

    ((equal '2 Method)             ; facet oriented input method
     (do ((data (read *input-stream* nil 'done) ; read all volumes into universe
               (setf data (read *input-stream* nil 'done))))
            ((atom data))
       (loop for terrain-segment in data        ; go through each volume segment
         do (setf (universe-volumes *universe*)
                  (append (make-volume-with-facet-data terrain-segment)
                          (universe-volumes *universe*))))))
    (t (terpri) (princ "Error: Method not implemented")
       (return-from set-up 'Done)))

  (close *input-stream*) (terpri)
  (princ ">Creation complete.") (terpri) (terpri)
  (setf *original-input-volumes* (universe-volumes *universe*))

                              ; complete initial setup functions
```

```lisp
   (find-all-ridges)
   (terpri)
   (make-convex-volumes)
   (setf *not-convex-volumes* 'nil)
   (setf *convex-volumes* (universe-volumes *universe*))
   (terpri) (terpri) (princ "Enter observer data now: ") (terpri) (terpri))


(defun print-header-1 ()
   (terpri)
   (terpri)
   (princ "*******************************************************************")
   (terpri)
   (princ "* Volume Creation and Display                          V1.1  *")
   (terpri)
   (princ "*******************************************************************")
   (terpri)
   (terpri))


;---------------------------INPUT METHOD ONE------------------------------------


(defun Initialize-volume (Volume Data)       ; expects data as:
   (cond                                      ; (line line line ....) where lines
are
      ((null Data) Volume)                    ; (point point)  where points are;
(x y z).
                                              ; (((x y z) (x y z)) ((x y z) (x y
z)))
      (t (create-new-line Volume (init-point (caar Data)) (init-point (cadar
Data)))
         (Initialize-volume Volume (cdr Data)))))


(defun create-new-line (Volume pt1 pt2)       ; put points and lines into volume
instance
   (pushnew pt1 (volume-points (eval Volume)))
   (pushnew pt2 (volume-points (eval Volume)))
   (pushnew (init-Line (init-vector '*origin*  pt1) (init-vector pt1 pt2))
                                      (volume-Edges (eval Volume)))))


;---------------------------INPUT METHOD TWO------------------------------------


(defun make-volume-with-facet-data (data)  ; construct a volume from a formatted
                                 ; list of data where format is:
                                 ; (facet facet facet...)
   (let ((terrain-facets (build-terrain data))
      (terrain-box (make-instance 'bounding-box))  ; used to find limits of data
                                          ; points
      (ground-volume (init-volume (list (list 'ground 'nil))))
      (air-volume (init-volume (list (list 'air 'nil))))
      (points-and-lines 'nil))

                      ; find all lines and points in terrain facets

      (setf points-and-lines (info-on-facets terrain-facets))
```

154

```lisp
                              ; assign values to air and ground volumes

    (setf (volume-composition (eval ground-volume)) 'ground)     ; set
composition
    (setf (volume-composition (eval air-volume)) 'air)
    (setf (volume-facets (eval ground-volume)) terrain-facets)   ; put terrain
facets
    (setf (volume-facets (eval air-volume)) terrain-facets)

                        ; construct top/bottom and sides of ground and air
volumes

    (send terrain-box :construct-bounding-box (first points-and-lines))
    (let ((point-1 (first (find-point (bounding-box-max-x terrain-box);corners
of terrain
                              (bounding-box-min-y terrain-box)
                              'nil
                              (first points-and-lines))))
      (point-2 (first (find-point (bounding-box-max-x terrain-box)
                              (bounding-box-max-y terrain-box)
                              'nil
                              (first points-and-lines))))
      (point-3 (first (find-point (bounding-box-min-x terrain-box)
                              (bounding-box-max-y terrain-box)
                              'nil
                              (first points-and-lines))))
      (point-4 (first (find-point (bounding-box-min-x terrain-box)
                              (bounding-box-min-y terrain-box)
                              'nil
                              (first points-and-lines))))
      (points-41 (stable-sort (find-point 'nil                  ; edges of terrain
                              (bounding-box-min-y terrain-box)
                              'nil
                              (first points-and-lines))
                        #'decreasing-sort-x-p))
      (points-12 (stable-sort (find-point (bounding-box-max-x terrain-box)
                              'nil
                              'nil
                              (first points-and-lines))
                        #'decreasing-sort-y-p))
      (points-23 (stable-sort (find-point 'nil
                              (bounding-box-max-y terrain-box)
                              'nil
                              (first points-and-lines))
                        #'decreasing-sort-x-p))
      (points-34 (stable-sort (find-point (bounding-box-min-x terrain-box)
                              'nil
                              'nil
                              (first points-and-lines))
                        #'decreasing-sort-y-p))
      (top-points 'nil)                                          ; top and bottom
      (bottom-points 'nil))                                      ; points

    (loop for P in (list point-1 point-2 point-3 point-4)        ; find top points
      do (setf top-points (adjoin (init-point (list
                                    (first
                                      (send (eval P) :list-format))
                                    (second
                                      (send (eval P) :list-format))
                                    *max-altitude*))
                            top-points)))
    (setf top-points (reverse top-points))
```

155

```lisp
(setf (volume-facets (eval air-volume))                          ; make top facet
  (adjoin (make-a-facet top-points)
          (volume-facets (eval air-volume)))))


(setf (volume-facets (eval air-volume))                          ; make top sides
  (adjoin (build-side-facet (fourth top-points)        ; and put in volume
                            (first top-points)
                            points-41)
          (volume-facets (eval air-volume))))
(setf (volume-facets (eval air-volume))
  (adjoin (build-side-facet (first top-points)
                            (second top-points)
                            points-12)
          (volume-facets (eval air-volume))))
(setf (volume-facets (eval air-volume))
  (adjoin (build-side-facet (third top-points)
                            (second top-points)
                            points-23)
          (volume-facets (eval air-volume))))
(setf (volume-facets (eval air-volume))
  (adjoin (build-side-facet (fourth top-points)
                            (third top-points)
                            points-34)
          (volume-facets (eval air-volume))))


(loop for P in (list point-1 point-2 point-3 point-4)   ;find bottom points
do (setf bottom-points (adjoin (init-point (list
                                 (first (send (eval P) :list-format))
                                 (second (send (eval P) :list-format))
                                 *min-altitude*))
                        bottom-points)))
(setf bottom-points (reverse bottom-points))
(setf (volume-facets (eval ground-volume))                       ; make bottom facet
  (adjoin (make-a-facet bottom-points)
          (volume-facets (eval ground-volume))))


(setf (volume-facets (eval ground-volume))                       ; make bottom sides
  (adjoin (build-side-facet (fourth bottom-points)     ; and put in volume
                            (first bottom-points)
                            points-41)
          (volume-facets (eval ground-volume))))
(setf (volume-facets (eval ground-volume))
  (adjoin (build-side-facet (first bottom-points)
                            (second bottom-points)
                            points-12)
          (volume-facets (eval ground-volume))))
(setf (volume-facets (eval ground-volume))
  (adjoin (build-side-facet (third bottom-points)
                            (second bottom-points)
                            points-23)
          (volume-facets (eval ground-volume))))
(setf (volume-facets (eval ground-volume))
  (adjoin (build-side-facet (fourth bottom-points)
                            (third bottom-points)
                            points-34)
          (volume-facets (eval ground-volume)))))


                        ; complete information on volumes

(setf points-and-lines (info-on-facets (volume-facets (eval air-volume))))
(setf (volume-points (eval air-volume)) (first points-and-lines))
(setf (volume-edges (eval air-volume)) (second points-and-lines))
```

```
      (setf points-and-lines (info-on-facets
                                  (volume-facets (eval ground-volume))))
      (setf (volume-points (eval ground-volume)) (first points-and-lines))
      (setf (volume-edges (eval ground-volume)) (second points-and-lines))
      (loop for V in (list air-volume ground-volume)
        do (let ()
        (terpri) (princ ">>>> Volume Created: ")
        (prin1 V) (princ " Composition: ")
        (prin1 (volume-composition (eval V)))))
      (list air-volume ground-volume)))

(defun decreasing-sort-x-p (A B)
  (cond ((> (first (send (eval A) :list-format))
        (first (send (eval B) :list-format))))))

(defun decreasing-sort-y-p (A B)
  (cond ((> (second (send (eval A) :list-format))
        (second (send (eval B) :list-format))))))



;*******************************************************************************
;
;              C O M P L E T E      S T A T I C      P H A S E
;              -----------------------------------------------------
;
;    Functions here complete the static phase of construction of the visibility
; regions.
;
; MAIN FUNCTIONS: SET-UP-2
;
;*******************************************************************************

(defun set-up-2 ()      ; finish initial setup (after observers created)
  (let ((observers (universe-observers *universe*)))
      (terpri) (terpri)
      (princ "------------PRE-PROCESS VISIBILITY INFORMATION------------")
      (terpri) (terpri)
      (loop for Obs in observers               ;divide up universe by visibilities
        do (make-visibility-regions Obs))
      (terpri) (terpri)
      (princ "Numeric errors: ") (prin1 *list-of-error-planes*)
      (terpri) (terpri)
      (send *universe* :save-static-items)
      (setf *final-visibility-regions* (universe-volumes *universe*))
      (setf *done-making-new-visibility-volumes-flag* 't)   ; speed things up
      (loop for Obs in observers                          ; find who can see what
        do (speed-demon)
        do (determine-visibility Obs))
      (terpri)(terpri)
      (princ "Determine Probability of Detection for visibility volumes")
      (terpri)
      (loop for V in (universe-volumes *universe*)    ; calc prob of detection for
        do (probabilities-assuming-independence-or V)) ; each volume
      (terpri) (terpri)
      (speed-demon)
      (connect-volumes)                                      ; build visibility graph
      (terpri)))
```

157

```
;;; -*- Mode:Common-Lisp; Base:10 -*-
;*****************************************************************
;;;      V I S I B I L I T Y    AND      R I D G E S
;;;
;;;    This file contains both the visibility determination code
;;; and the ridge creation and initial air-volume "convexizing"
;;; code.  The visibility code is first, followed by the ridge
;;; code.
;;;
;;; THESIS                                   D.H. Lewis 10/11/86
;*****************************************************************
;;;
;;; VISIBILITY REGIONS         D.H. Lewis              10 Aug 88
;;;
;;;              ---------------------------------------
;;;
;;;     Contains the Observer flavor; code for creating and
;;; manipulating observer data; code for making visibility
;;; visibility regions; code for determining the visibility of
;;; visibility volumes; and finally code for finding the probability
;;; of detection for the visibility volumes.
;;;
;;; Main functions:   MAKE-VISIBILITY-REGIONS (OBSERVER)
;;;                   DETERMINE-VISIBILITY (OBSERVER)
;;;                   INIT-OBSERVER (COORDINATES EFFECTIVNESS)
;;;                   CONNECT-VOLUMES ()
;;;                   DETERMINE-VISIBILITY-1
;;;                   DETERMINE-VISIBILITY-2
;;;
;;; Other functions: MAKE-OBSERVER-NAME
;;;                   COLINEAR-P
;;;                   FIND-T
;;;                   PROBABILITIES-ASSUMING-INDEPENDENCE-OR
;;;                   PROBABILITIES-ASSUMING-INDEPENDENCE-AND
;;;                   CLEAR-VISIBILITY
;;;                   MATCH-FACET-WITH-ANOTHER-VOLUME
;;;                   SHOW-CONNECTIVITY
;;;                   CLEAR-CONNECTIVITY
;;;                   CONNECTIVITY-METRIC
;;;                   FINE-IF-VISIBILITY-LINE-BLOCKED-P
;;;                   SET-ARITHMETIC-CENTERS
;;;                   SET-ZERO-PD
;;;
;;;         **-ALSO CONTAINS RIDGE FUNCTIONS
;*****************************************************************

(defvar *list-of-observers* 'nil)
(defvar *observer-counter* '0)


;----------------------------------------------------------------
;    FLAVORS USED TO CREATE OR MANIPULATE VISIBILITY REGIONS
;----------------------------------------------------------------

(defflavor Observer
        (Effectivness
      Position)
        (graphic)        ; for display
    :gettable-instance-variables
    :settable-instance-variables
    :inittable-instance-variables
    :outside-accessible-instance-variables)
```

```
;-------------METHODS FOR OBSERVERS-----------------

(defmethod (observer :make-node-list) ()
  (list (reverse (append (list '1) (reverse (send (eval position)
:list-format))))))

(defmethod (observer :make-polygon-list) ()
  '((0 0)))


;-----------FUNCTIONS FOR OBSERVERS----------------

(defun make-observer-name ()
  (gensym (incf *observer-counter*))
  (intern (gensym "observer")))


(defun init-observer (coord effectivness)
  (let* ((temp (make-observer-name))
      (position (init-point coord))
      (volume-location (locate-point-air position)))
                                          ;which air volumes contain obs?
      (cond ((null volume-location)              ;make sure not underground
        (terpri)
        (princ "Invalid location for observer (underground)") (terpri)
        (return-from init-observer 'nil)))
      (set temp (make-instance 'Observer
                        :Effectivness effectivness
                        :Position position))
      (pushnew temp *list-of-observers*)
      (setf (universe-observers *universe*) (adjoin temp
                                      (universe-observers *universe*))))
      temp))

;-------------------------------------------------------------
; Determine all observer planes, and make visibility regions
;-------------------------------------------------------------

(defun make-visibility-regions (observer)
  (let ((ground-volumes 'nil)
    (air-volumes 'nil)
    (ridges 'nil)
    (planes 'nil)
    (result-volume-list 'nil))
                              ; find all air,ground volumes, visible ridges
    (terpri) (terpri)
    (princ "making visibility regions for: ")
    (prin1 observer) (terpri) (terpri)
    (loop for V in (universe-volumes *universe*)
      do (cond ((equal 'ground (volume-composition (eval V)))
            (setf ground-volumes (adjoin V ground-volumes))
            (loop for L in (volume-edges (eval V))
              do (cond ((equal 'ridge (line-segment-characteristics (eval L)))
                    (cond ((not (colinear-p
                              (observer-position (eval observer))
                                    L))
                          (setf ridges (adjoin L ridges)))))))
            (t (setf air-volumes (adjoin (list V) air-volumes))
              (setf (universe-volumes *universe*)
                  (remove V (universe-volumes *universe*))))))
                              ; make all visibility limiting planes
    (loop for R in ridges
```

159

```
              do (setf planes
                      (adjoin (make-a-plane (observer-position (eval Observer)) R)
                              planes)))
                                      ; intersect all air volumes with planes
        (princ "Air volumes: ") (prin1 air-volumes) (terpri)
        (princ "Limiting planes of visibility: ") (prin1 planes) (terpri) (terpri)
        (setf result-volume-list (intersect-all-planes-with-volumes planes
                                                   air-volumes))
        (loop for V in result-volume-list
          do (push (car V) (universe-volumes *universe*))))
      (send *universe* :save-static-items)      ; save the state of the static model
      (universe-volumes *universe*))

(defun colinear-p (point line)
  (let ((tx (find-t '0 point line))                   ; find x,y,z t parameters
        (ty (find-t '1 point line))
        (tz (find-t '2 point line))
        (t-list 'nil)
        (t-list-reduced 'nil))
      (setf t-list (substitute '0.0 'nil (list tx ty tz)))
      (setf t-list-reduced (remove 'nil (list tx ty tz)))
      (cond ((equal '1 (length t-list-reduced))
               (return-from colinear-p
             (apply 'and (mapcar 'equal-error (send (eval point) :list-format-real)
                                              (send (eval line) :backsubs t-list)))))
            ((equal '2 (length t-list-reduced))
             (return-from colinear-p (apply 'equal-error t-list-reduced)))
            (t (return-from colinear-p (and (equal-error tx ty)
                                            (equal-error tx tz)))))))

(defun find-t (nr point line)
  (let ((denom (nth nr (send (eval (line-segment-direction-vector
                                  (eval line))) :list-format)))
        (numerator (- (nth nr (send (eval point) :list-format))
                      (nth nr (send (eval (line-segment-position-vector
                                  (eval line))) :list-format)))))
      (cond ((equal-zero-p denom)
             (return-from find-t 'nil))
            (t (return-from find-t (/ numerator denom))))))


;-----------------------------------------------------------------------
; Determine visibility of visibility regions
;-----------------------------------------------------------------------

(defun determine-visibility (observer)
  (determine-visibility-1 observer))

(defun determine-visibility-1 (observer)
                                 ; determine the visibility status (yes or no)
                                 ; of all air volumes w/ respect to a sigle
observer
                                 ; using a fast method
  (terpri) (terpri)
  (princ "Visibility determination for: ") (prin1 observer)
  (terpri) (terpri)
  (let ((ground-volumes 'nil)
        (air-volumes 'nil)
        (ground-facets 'nil)
        (volumes-containing-observer
          (locate-point-air (observer-position (eval observer)))))
```

160

```
                              ; find all air,ground volumes, and ground facets
                              ; make bounding boxes for ground facets

  (set-arithmetic-centers)
  (loop for V in volumes-containing-observer
    do (princ "   ")
    do (prin1 V)
    do (princ " visible")
    do (terpri))
  (loop for V in (universe-volumes *universe*)
    do (cond ((equal 'air (volume-composition (eval V)))
          (cond ((not (member-p V volumes-containing-observer))
                (setf air-volumes (adjoin V air-volumes)))))
        (t (setf ground-volumes (adjoin V ground-volumes))
          (loop for F in (volume-facets (eval V))
            do (setf ground-facets (adjoin F ground-facets)))))))

                                  ; build bounding box for ground facets

  (loop for F in ground-facets
    do (send (eval F) :construct-bounding-box (send (eval F) :points)))

                              ; determine visibility of all air volumes
                              ; containg the observer

  (loop for V in volumes-containing-observer
    do (setf (volume-visibility (eval V))
            (adjoin observer (volume-visibility (eval V))))))

                          ; determine visibility of remainder of air volumes
                          ; by seeing if visibility line intersects a ground
                          ; facet

  (loop for V in air-volumes
    do (let ((visibility-line (make-line (observer-position (eval observer))
                              (volume-arithmetic-center (eval V))))
          (blocked-flag 'nil))
      (loop for F in ground-facets
        do (let ((facet-plane (init-plane (send (eval F) :list-coeff)))
              (I 'nil))
          (cond ((subs-line-into-plane-equation visibility-line
                                    facet-plane))
              ((not blocked-flag)
                (cond ((send (eval visibility-line) :strattle-plane-p
                                          facet-plane)
                      (setf I (find-intercept-point facet-plane
                                      visibility-line))
                      (cond ((send (eval F) :inside-bounding-box-p I)
                            (cond ((inside-facet-p I F)
                                      (princ "   ") (prin1 V)
                                  (princ " not visible") (terpri)
                                  (setf blocked-flag 't)))))))))))
      (cond ((not blocked-flag)
          (princ "   ") (prin1 V) (princ " visible") (terpri)
          (setf (volume-visibility (eval V))
              (adjoin observer (volume-visibility (eval V)))))))))
  (terpri)
'nil))
```

161

```lisp
(defun determine-visibility-2 (observer)
                            ; determine the visibility status (yes or no)
                            ; of all air volumes w/ respect to a sigle observer
                            ; using a slow method
  (terpri) (terpri)
  (princ "Visibility determination for: ") (prinl observer)
  (terpri) (terpri)
  (let ((ground-volumes 'nil)
    (air-volumes 'nil)
    (ground-facets 'nil)
    (volumes-containing-observer
      (locate-point-air (observer-position (eval observer)))))
    (set-arithmetic-centers)

                                ; determine visibility of all air volumes
                                ; containg the observer

    (loop for V in volumes-containing-observer
      do (setf (volume-visibility (eval V))
           (adjoin observer (volume-visibility (eval V)))))
    (loop for V in volumes-containing-observer
      do (princ "  ")
      do (prinl V)
      do (princ " visible")
      do (terpri))

                                    ; find who rest of volumes are, and make list
                                    ; of blocking ground facets.  Remove all
                                    ; vertical ground facets.

    (loop for V in (universe-volumes *universe*)
      do (cond ((equal 'air (volume-composition (eval V)))
            (cond ((not (member-p V volumes-containing-observer))
                (setf air-volumes (adjoin V air-volumes)))))
           (t (setf ground-volumes (adjoin V ground-volumes))
             (loop for F in (volume-facets (eval V))
               do (cond ((and (member-p '0 (send (eval F) :list-coeff-3))
                         (> 2 (length (remove
                                       '0
                                       (send (eval F) :list-coeff-3))))))
                    (t (setf ground-facets (adjoin F ground-facets)))))))))
    (setf ground-facets (remove-duplicates ground-facets))
    (loop for F in ground-facets
      do (send (eval F) :construct-bounding-box (send (eval F) :points)))
                                ; determine visibility of remainder of air volumes
                                ; by seeing if visibility line intersects a ground
                                ; facet
    (loop for V in air-volumes
      do (let ((visibility-line (make-line (observer-position (eval observer))
                                (volume-arithmetic-center (eval V)))))
        (cond ((find-if-visibility-line-blocked-p visibility-line
                                                  ground-facets
                                                  ground-volumes)
              (princ "  ") (prinl V)
              (princ " not visible") (terpri))
            (t (princ "  ") (prinl V) (princ " visible") (terpri)
              (setf (volume-visibility (eval V))
                (adjoin observer (volume-visibility (eval V))))))))
    'nil))
```

```lisp
(defun find-if-visibility-line-blocked-p (visibility-line
                                          ground-facets
                                          ground-volumes)
  (loop for F in ground-facets
    do (let ((intercept-point (find-intercept-point
                                 (init-plane (send (eval F) :list-coeff))
                                 visibility-line))
             (location-volumes 'nil))
      (cond ((null intercept-point)
               (return-from find-if-visibility-line-blocked-p 'nil))
              ((not (send (eval F) :inside-bounding-box-p intercept-point))
               (return-from find-if-visibility-line-blocked-p 't))
              (t (setf location-volumes (locate-point intercept-point))
                 (loop for V in ground-volumes
                       do (cond ((member-p V location-volumes)
                                 (return-from find-if-visibility-line-blocked-p 't))))
                 (return-from find-if-visibility-line-blocked-p 'nil)))))))


(defun probabilities-assuming-independence-or (volume)
                            ; set volume probability of detection using an
                            ; asssumption of indepedence between observers,
                            ;  and an "or" combination technique
  (let ((temp '1.0))
    (terpri)
    (prin1 volume) (princ " has P.D.: ")
    (cond ((not (null (volume-visibility (eval volume))))
       (loop for Obs in (volume-visibility (eval volume))
         do (setf temp (* temp (- '1.0 (observer-effectivness (eval Obs))))))
       (setf (volume-probability-of-detection (eval volume)) (- '1.0 temp))
       (prin1 (- '1.0 temp)))
      (t (setf (volume-probability-of-detection (eval volume)) '0.0)
         (prin1 '0.0)))))

(defun probabilities-assuming-independence-and (volume)
                            ; set volume probability of detection using an
                            ; asssumption of indepedence between observers,
and
                            ; an "and" combination technique
  (let ((temp '1.0))
    (terpri)
    (prin1 volume) (princ " has P.D.: ")
    (cond ((not (null (volume-visibility (eval volume))))
       (loop for Obs in (volume-visibility (eval volume))
         do (setf temp (* temp (observer-effectivness (eval Obs)))))
       (setf (volume-probability-of-detection (eval volume)) temp)
       (prin1 temp))
      (t (setf (volume-probability-of-detection (eval volume)) '0.0)
         (prin1 '0.0)))))

(defun set-arithmetic-centers ()
  (loop for V in (universe-volumes *universe*)
    do (setf (volume-arithmetic-center (eval V)) (send (eval V)
:find-arithmetic-center))))


(defun clear-visibility ()                       ; clear out observer visibility
info
  (loop for V in (universe-volumes *universe*)
    do (setf (volume-probability-of-detection (eval V)) 'nil)
    do (setf (volume-visibility (eval V)) 'nil))
  'Done)
```

163

```
(defun set-zero-PD ()                                    ; set all air volume PD's to
zero
  (loop for V in (universe-volumes *universe*)
    do (cond ((equal 'air (volume-composition (eval V)))
          (setf (volume-probability-of-detection (eval V)) '0.0))))
  'done)


;***************************************************************************
;
;                       C O N N E C T I V I T Y
;
;***************************************************************************


;-------------------------------------------------------
;         Connectivity between volumes
;-------------------------------------------------------

(defun Connect-volumes ()       ; connect all air volumes by facets.
  (let ((volumes (universe-volumes *universe*)))
    (terpri)
    (terpri) (princ "Connecting volumes:") (terpri) (terpri)
    (loop for V in volumes
      do (prin1 V)
      do (princ " Connected to: ")
      do (setf (volume-connected-to (eval V)) 'nil)
      do (cond ((equal 'air (volume-composition (eval V)))
          (loop for F in (volume-facets (eval V))
            do (send (eval F) :find-facet-center)
            do (send (eval F) :add-volume-to-left-connects V)
            do (let ((match (match-facet-with-another-volume F V)))
                (cond ((and
                       (not (null match))
                        (not (equal 'ground (volume-composition (eval
match)))),)
                      (send (eval F) :add-volume-to-right-connects match))
                     ((null match)
                      (let* ((volumes (locate-point-air
                                   (facet-center (eval F)))))
                       (loop for Connect-vol in (remove V volumes)
                         do (send (eval F)
                                 :add-volume-to-right-connects Connect-vol)
                       ))))))))
      (loop for F in (volume-facets (eval V))
    do (setf (volume-connected-to (eval V))
          (append (second (facet-connects (eval F)))
                 (volume-connected-to (eval V)))))
      (setf (volume-connected-to (eval V))
       (remove-duplicates (volume-connected-to (eval V))))
      (setf (volume-connected-to (eval V))
       (remove 'nil (volume-connected-to (eval V))))
      (setf (volume-connected-to (eval V))
       (remove V (volume-connected-to (eval V))))
      (loop for V2 in (volume-connected-to (eval V))       ; remove ground volumes
    do (cond ((equal 'ground (volume-composition (eval V2)))
          (setf (volume-connected-to (eval V))
              (remove V2 (volume-connected-to (eval V)))))))
      (prin1 (volume-connected-to (eval V)))
      (terpri))
    (terpri)))
```

```lisp
(defun match-facet-with-another-volume (Facet Volume)
                                ; return the name of the unique facet which is
shared
                                ; between two volumes, else return NIL. Volume is
                                ; assumed to contain facet
  (let ((volumes (universe-volumes *universe*)))
    (loop for V in volumes
      do (cond ((not (equal V Volume))
          (cond ((member-p Facet (volume-facets (eval V)))
                  (return-from match-facet-with-another-volume V))
                ((or (member-p V (second (facet-connects (eval Facet))))
                    (member-p V (first (facet-connects (eval Facet)))))
                  (return-from match-facet-with-another-volume V))))))
    'nil))

(defun show-connectivity ()                      ; show how volumes connect
  (terpri)
  (loop for V in (universe-volumes *universe*)
    do (let ()
    (terpri) (prin1 V)
    (princ " <-> ")
    (prin1 (volume-connected-to (eval V))))))


(defun clear-connectivity ()                     ; clear state of
connectivity
  (loop for V in (universe-volumes *universe*)
    do (setf (volume-connected-to (eval V)) 'nil))
  'done)

(defun connectivity-metric ()
  (terpri)
  (loop for V in (universe-volumes *universe*)
    do (prin1 V)
    do (princ ":  Connections: ")
    do (prin1 (length (volume-connected-to (eval V))))
    do (princ "  Facets: ")
    do (prin1 (length (volume-facets (eval V))))
    do (cond ((or (equal (length (volume-connected-to (eval V)))
                    (1- (length (volume-facets (eval V)))))
            (equal (length (volume-connected-to (eval V)))
                  (length (volume-facets (eval V)))))
        (t (princ "  -- possible error")))
    do (terpri)))
```

165

```
;***********************************************************************
;;;
;;;            RIDGE CREATION AND MANIPULATION FUNCTIONS
;;;   D.H. LEWIS                                         22May88
;;;
;;;        -----------------------------------------------
;;;
;;;   Functions to find, make, and manipulate ridge lines.
;;;
;;;   Main functions:  FIND-ALL-RIDGES ()
;;;                    LINE-IS-A-RIDGE-P (LINE VOLUME)
;;;                    MAKE-CONVEX-VOLUMES ()
;;;
;;;   Other functions: FIND-FACETS-WHICH-CONTAIN-EDGE
;;;                    PUT-FACET-ON-CORRECT-SIDE
;;;                    FIND-OVERLAPPING-FACETS
;;;                    FIND-HIGHEST-FACET
;;;                    RIDGE-LENGTH-SORT
;;;
;***********************************************************************

;----Make ridges----

(defun find-all-ridges ()    ; look for line-segments which are ridges
  (terpri) (terpri)
  (princ "Find all ridges in ground terrain: ") (terpri) (terpri)
  (loop for Volume in (universe-volumes *universe*)
     do (cond ((equal 'ground (volume-composition (eval Volume)))
              (loop for E in (Volume-edges (eval Volume))
                   do (princ "Ridge check, line: ")
               do (prinl E)
               do (cond ((line-is-a-ridge-p E Volume)
                         (setf (line-segment-characteristics (eval E))
                           'ridge)
                         (princ " --  Ridge")
                       (terpri))
                       (t (setf (line-segment-characteristics (eval E))
                            'nil)
                       (terpri)))))))))

(defun line-is-a-ridge-p (Line Volume)  ; T if line is a ridge
  (let ((Facets (find-facets-which-contain-edge Line Volume))
    (Edge-vertical-plane (make-vertical-plane Line))
    (Right-side-facets 'nil)
    (Highest-right-side-facet 'nil)
    (Left-side-facets 'nil)
    (Highest-left-side-facet 'nil)
    (Vertical-facets 'nil)
    (Overlapping-facets 'nil))

                               ; divide facets into left and right halves based
                               ; on spacial relationship of middle point
                               ; with vertical plane of Line

    (loop for F in facets
       do (setf (get F 'center) (init-point (mean-point-in-facet F )))
       do (setf (get F 'opposite-points) 'nil)
       do (let ((side (put-facet-on-correct-side F Edge-vertical-plane)))
          (cond ((not (null (first side)))
                 (setf Left-side-facets (adjoin (first side) Left-side-facets)))
                ((not (null (second side)))
                 (setf Vertical-facets (adjoin (second side) Vertical-facets)))
```

166

```lisp
        ((not (null (third side)))
          (setf Right-side-facets (adjoin (third side)
Right-side-facets))))))

                                        ; do not consider vertical facets in any manner

    (cond ((not (null Vertical-facets))
       (return-from Line-is-a-ridge-p 'nil)))
                        ; handle overlapping facets by creating a new facet center
                        ; composed of average of facet points on correct side of
                        ; possible ridge line

    (cond ((or (null Left-side-facets)
           (null Right-side-facets))
       (cond ((null Left-side-facets)
          (setf Overlapping-facets
             (find-overlapping-facets Edge-vertical-plane
                                   Right-side-facets))
          (loop for F in Overlapping-facets
            do (setf Right-side-facets (remove F Right-side-facets))))
          ((null Right-side-facets)
           (setf Overlapping-facets
              (find-overlapping-facets Edge-vertical-plane
                                    Left-side-facets))
           (loop for F in Overlapping-facets
                do (setf Left-side-facets (remove F Left-side-facets)))))
       (cond ((null Overlapping-facets)              ; have an internal facet
          (return-from line-is-a-ridge-p 'nil)))
       (loop for F in Overlapping-facets
          do (setf (get F 'center) (init-point (average-of-points
                                       (get F 'opposite-points))))
          do (let ((side (put-facet-on-correct-side F Edge-vertical-plane)))
               (cond ((not (null (first side)))
                  (setf Left-side-facets
                     (adjoin (first side) Left-side-facets)))
                 ((not (null (second side)))
                  (setf Vertical-facets
                     (adjoin (second side) Vertical-facets)))
                 ((not (null (third side)))
                  (setf Right-side-facets
                     (adjoin (third side) Right-side-facets))))))))
    ; reduce lists of left- and right- facets to one facet
    ; per side, based upon z-value of mean point of facet
    (cond ((< 1 (length Left-side-facets))
       (setf Highest-left-side-facet (find-highest-facet Left-side-facets)))
      (t (setf Highest-left-side-facet (first Left-side-facets))))
     (cond ((< 1 (length Right-side-facets))
       (setf Highest-right-side-facet (find-highest-facet Right-side-facets)))
      (t (setf Highest-right-side-facet (first Right-side-facets))))
                        ; find if line is a ridge by subs right side mean value
                        ; into left-side plane equation.  If resultant Z value
                        ; is greater than right-side mean value z-value, have
                        ; a ridge, else not

    (let* ((point (send (eval (get Highest-right-side-facet 'center))
:list-format))
       (z-right-point-into-left-plane
         (send (eval Highest-left-side-facet)
            :find-z-given-xy (first point) (second point))))
       (cond ((> z-right-point-into-left-plane (third point))
         (return-from line-is-a-ridge-p 't))
         (t (return-from line-is-a-ridge-p 'nil))))))
```

```
(defun find-facets-which-contain-edge (Edge Volume)
  (let ((temp 'nil))
    (loop for F in (volume-facets (eval Volume))
        do (cond ((member-p Edge (facet-edges (eval F)))
            (setf temp (adjoin F temp)))))
    temp))

(defun put-facet-on-correct-side (Facet Plane )
  (let* ((Ao (fourth (send (eval plane) :list-coeff)))
        (Ao-Point (subs-point-into-equation (send (eval plane) :list-coeff-3)
                                            (get Facet 'center)))
        (Left 'nil)
        (Vertical 'nil)
        (Right 'nil))
    (cond ((GT Ao Ao-point)
      (pushnew Facet Left))
     ((LT Ao Ao-point)
      (pushnew Facet Right))
     (t (pushnew Facet Vertical)))
    (list (first Left) (first Vertical) (first Right))))


(defun find-overlapping-facets (Vertical-plane Facets)
  (let* ((Line-Ao (fourth (send (eval vertical-plane) :list-coeff)))
     (Facet-center-Ao 'nil)
     (overlapping-facets 'nil))
    (loop for F in Facets
      do (setf facet-center-Ao (send (eval Vertical-plane)
:subs-point-into-plane
                         (get F 'center)))
      do (loop for P in (send (eval F) :points)
      do (let ((Point-Ao
                 (send (eval Vertical-plane) :subs-point-into-plane P)))
          (cond ((or (and (GT Line-Ao Point-Ao)
                          (LT Line-Ao Facet-center-Ao))
                    (and (LT Line-Ao Point-Ao)
                         (GT Line-Ao Facet-center-Ao)))
                (setf overlapping-facets (adjoin F overlapping-facets))
                (setf (get F 'opposite-points)
                    (adjoin P (get F 'opposite-points)))))))))
    overlapping-facets))


(defun find-highest-facet (List-of-facets)
  (let ((highest-z (third
                 (send (eval (get (first list-of-facets) 'center)) :list-format)))
    (highest-facet (first List-of-facets)))
    (loop for F in (rest List-of-facets)
    do (let ((z (third (send (eval (get F 'center)) :list-format))))
        (cond ((GT z highest-z)
               (setf highest-z z)
               (setf highest-facet F)))))
    highest-facet))



;----------------------------------------------
;---Use ridges to make convex air volumes----
;----------------------------------------------
```

168

```lisp
(defun make-convex-volumes ()              ; intersect all vertical planes from ridge
  (let ((air-volume-list '())              ; line-segments with all volume(s).
        (volume-list 'nil)                 ; Makes all air volumes convex,
        (ridge-list 'nil)                  ; guarenteed.
        (plane-list 'nil))

    (terpri) (terpri)
    (princ "Making air volumes convex:")
    (terpri) (terpri)

                                 ; seperate all air and ground volumes
                                 ; and find ridge lines

    (loop for V in (Universe-volumes *universe*)
     do (cond ((equal 'air (volume-composition (eval V)))
              (setf air-volume-list (adjoin (list V) air-volume-list))
              (loop for E in (volume-edges (eval V))
                 do (cond ((equal 'ridge
                                  (line-segment-characteristics (eval E)))
                          (setf ridge-list (adjoin E ridge-list)))))
              (setf (universe-volumes *universe*)
                    (remove V (universe-volumes *universe*))))))

                 ; reduce list of ridge lines, and construct vertical planes
                 ; for them.  ridges are sorted by length, longest first

    (setf ridge-list (remove-duplicates ridge-list))
    (setf ridge-list (remove 'nil ridge-list))
    (setf ridge-list (stable-sort ridge-list #'ridge-length-sort))
    (loop for R in ridge-list
      do (setf plane-list (adjoin (make-vertical-plane R) plane-list)))
    (setf plane-list (reverse plane-list))
    (princ "Air volumes: ") (prin1 air-volume-list) (terpri)
    (princ "Ridge planes: ") (prin1 plane-list) (terpri) (terpri)

                        ; intersect all ridge planes with all air volumes

    (setf volume-list (intersect-all-planes-with-volumes plane-list
                                          air-volume-list))

                        ; update universe with new volumes created

    (loop for V in volume-list
      do (push (car V) (universe-volumes *universe*)))
    (universe-volumes *universe*)))


(defun ridge-length-sort (A B)                     ;return T iff A > B
  (> (send (eval A) :length)
     (send (eval B) :length)))
```

169

```
;;; -*- Mode: LISP; Syntax: Common-lisp -*-
;*************************************************************
;;;
;;; D.H. Lewis              CS4452/THESIS              5May88
;;;
;*************************************************************
;
;                        FLAVORS AND METHODS
;
;       ---------------------------------------------
;
; FLAVOR:    .....................Point
;
; METHODS: :LIST-FORMAT              ; give the x,y and z values as a
three-tuple
;          :LIST-FORMAT-REAL         ; same, in real number format
;          :LIST-FORMAT-4            ; give agraphics 4-tuple "(x y z 1)"
;          :PRINT                    ; print info on point
;
; FLAVOR:    ....................Vector
;
; METHODS: :LENGTH                   ; calculate length of vector
;          :UNIT-VECTOR              ; return the coeff of the unit vector
;          :ENDPOINTS                ; give the names of the endpoints of the
vector
;          :LIST-FORMAT              ; give the coeffs of the vector as a
3-tuple
;          :LIST-FORMAT-REAL         ; same, execept with real numbers
;          :PRINT                    ; print coeff values to output file
;
; FLAVOR:    ....................Line-segment
;
; METHODS: :ENDPOINTS               ; Return the endpoints of the line-segment
;          :ENDPOINT-LIST           ; Return endpoints as explicit 4-tuples
;          :OTHER-END (ENDPOINT)    ; Given one endpoint, return the other
;
;          :START-POINT             ; Return the start point of the
line-segment
;          :END-POINT               ;   "      "  end point    "    "        "
;          :LENGTH                  ; Find and return the length of the
;                                   ;     line-segment
;          :BACKSUBS (T-LIST)       ; Subsitute the (Tx Ty Tz) list into
;                                   ;     the line equation
;          :MID-POINT               ; Find the mid point of the line-segment
;          :STRATTLE-PLANE-P (PLANE)   ; do the endpoints of the line-segment
;                                      lie on opposite sides of the plane?
;          :PRINT
;
; FLAVOR:    ....................Plane
;
; METHODS: :TEST-EQUAL (PLANE)      ; Do two planes have the same coeffs?
;          :LIST-COEFF              ; List 4-tuple (X Y Z Ao) for plane
;          :LIST-COEFF-3                ; List 3-tuple (X Y Z) for plane
;          :SUBS-POINT-INTO-PLANE (POINT) ; Get Ao coeff from X,Y,Z values of
point
;          :FIND-Z-GIVEN-XY (X Y)   ; Return Z value of point in plane
;          :FIND-Y-GIVEN-XZ (X Z)   ;   "    X  "   "    "     "   "    "
;          :FIND-X-GIVEN-YZ (Y Z)   ;   "    Y  "   "    "     "   "    "
;          :PRINT
;
; FLAVOR:    ....................Bounding-box
;
```

170

```
; METHODS:    :CONSTRUCT-BOUNDING-BOX (POINTS)  ; Build a 3-D limits for list of
points
;             :INSIDE-BOUNDING-BOX (POINT)      ; Is the point inside the limits?
;
; FLAVOR:     ...................Facet
;
; METHODS:    :POINTS
;             :PRINT
;
; FLAVOR:     ...................Volume
;
; METHODS:    :MAKE-EQUAL
;             :CLEAR
;             :FIND-ARITHMETIC-CENTER
;             :MAKE-NODE-LIST
;             :MAKE-POLYGON-LIST
;             :PRINT
;
; FLAVOR:     ...................Universe
;
; METHODS:    :SAVE-STATIC-ITEMS
;             :REVERT-STATIC-ITEMS
;*****************************************************************
;
; OTHER FUNCTIONS: MAKE-ORIGIN            INIT-POINT
;                  MAKE-NULL-VECTOR       INIT-NEW-POINT
;                  MAKE-POINT-NAME        INIT-VECOTR
;                  MAKE-LINE-NAME         INIT-NEW-VECTOR
;                  MAKE-VECTOR-NAME       INIT-LINE
;                  MAKE-FACET-NAME        INIT-NEW-LINE
;                  MAKE-PLANE-NAME        INIT-PLANE
;                  MAKE-VOLUME-NAME       INIT-NEW-PLANE
;                  MAKE-ALL-FACETS        INIT-VOLUME
;                  MAKE-NEW-FACET         INIT-FACET-2
;                  MAKE-A-FACET
;
;                  FIND-OR-MAKE-LINE
;                  OLD-LINE-DV
;                  INITIALIZE-SEARCH
;                  SEARCH-TO-MAKE-FACET
;                  BUILD-SIDE-FACET
;                  BUILD-TERRAIN
;
;*****************************************************************
(defvar *origin*)
(defvar *null-vector*)
(defvar *one-vector* '(1.0 1.0 1.0 1.0))
(defvar *one-vector-3* '(1.0 1.0 1.0))
(defvar *zero-vector* '(0.0 0.0 0.0 0.0))
(defvar *zero-vector-3* '(0.0 0.0 0.0))
(defvar *max-counter-value* '9999)
(defvar *done-making-new-visibility-volumes-flag* 'nil)

(defvar *list-of-points* 'nil)
(defvar *points-counter* '0)
(defvar *minimum-points-counter* '0)

(defvar *list-of-vectors* 'nil)
(defvar *vectors-counter* '0)
(defvar *minimum-vectors-counter* '0)

(defvar *list-of-lines* 'nil)
```

171

```lisp
(defvar *lines-counter* '0)
(defvar *minimum-lines-counter* '0)

(defvar *list-of-planes* 'nil)
(defvar *planes-counter* '0)
(defvar *minimum-planes-counter* '0)

(defvar *list-of-facets* '())
(defvar *facets-counter* '0)
(defvar *minimum-facets-counter* '0)

(defvar *list-of-volumes* '())
(defvar *volumes-counter* '0)
(defvar *minimum-volumes-counter* '0)


;------POINT------

(defflavor point
    (x-coord
     y-coord
     z-coord)
    ()
    :gettable-instance-variables
    :settable-instance-variables
    :inittable-instance-variables
    :outside-accessible-instance-variables)

(defmethod (point :List-format) ()   ; return a 3-tuple "(X Y Z)"
  (list x-coord y-coord z-coord))

(defmethod (point :List-format-real) () ; return a real valued 3-tuple
  (map 'list '* (list x-coord y-coord z-coord)
       (make-list 3 :initial-element '1.0)))

(defmethod (point :List-format-4) ()   ; return list in graphics format
  (list x-coord y-coord z-coord '1))

(defmethod (point :print) ()
  (pprint (list x-coord y-coord z-coord) *output-stream*))



;-------VECTOR------

(defflavor vector
    (i
     j
     k
     Start-point
     End-point)
    ()
    :gettable-instance-variables
    :settable-instance-variables
    :inittable-instance-variables
    :outside-accessible-instance-variables)

(defmethod (vector :length) ()   ; Calculate the length of a vector
  (sqrt (abs (+ (* i i) (* j j) (* k k)))))
```

172

```
(defmethod (vector :unit-vector) ()   ; make a unit vector from a vector
  (let ((vector-length (send self :length)))
    (cond ((equal-zero-p vector-length) '(0.0 0.0 0.0))
          (t (map 'list '/ (send self :list-format)
             (make-list 3 :initial-element vector-length))))))

(defmethod (vector :endpoints) ()   ; find the endpoints of the vector
  (list Start-point End-point))

(defmethod (vector :list-format) ()   ; return the values of the
                                      ;   vector as a 3-tuple
  (list i j k))

(defmethod (vector :list-format-real) ()   ; return a real valued 3-tuple
  (map 'list '* (list i j k) (make-list 3 :initial-element '1.0)))

(defmethod (vector :print) ()
  (pprint (list i j k Start-point End-point) *output-stream*))

;--------LINE SEGMENT-------

(defflavor line-segment                ; position vector can point to either end of
    (t-max                   ; direction vector. direction vector can point
     position-vector         ; in either direction between endpoints
     direction-vector
     characteristics)        ; ridge, valle·  etc
    ()
    :gettable-instance-variables
    :settable-instance-variables
    :inittable-instance-variables
    :outside-accessible-instance-variables)

(defmethod (line-segment :endpoints) ()    ;get endpoints of the line segment
  (send (eval direction-vector) :endpoints))

(defmethod (line-segment :endpoint-list) ()   ; get endpoints in graphics format
  (list (send (eval (car (send self :endpoints))) :list-format-4)
        (send (eval (cadr (send self :endpoints))) :list-format-4)))

(defmethod (line-segment :other-end) (endpoint)
                                 ; find the endpoint of the line-segment
                                 ; opposite of the given endpoint
  (let ((line-endpoints (send self :endpoints)))
    (cond ((equal endpoint (first line-endpoints))
           (second line-endpoints))
          (t (first line-endpoints)))))

(defmethod (line-segment :start-point) ()
                             ; what is the start point of the line-segment?
  (vector-start-point (eval direction-vector)))

(defmethod (line-segment :end-point) ()
                             ; what is the end point of the line segment?
  (vector-end-point (eval direction-vector)))

(defmethod (line-segment :length) ()   ; how long is the line-segment?
  (send (eval direction-vector) :length))

(defmethod (line-segment :backsubs) (t-list)   ; subs a list of t-parameters
                                     ; back into the line equation to get
                                     ; the (x y z) coord of the point
  (mapcar '+ (send (eval position-vector) :list-format-real)
```

173

```lisp
                 (mapcar '* t-list
                         (send (eval direction-vector) :list-format-real))))

     (defmethod (line-segment :midpoint) ()
       (let ((t-half (/ t-max '2.0)))
         (send self :backsubs (list t-half t-half t-half))))


     (defmethod (line-segment :strattle-plane-p) (plane)
                                            ; return T iff the endpoints of self
                                            ; are on opposite sides of the given
plane
       (let ((Ao-1 (send (eval plane) :point-into-equation
                                       (first (send self :endpoints))))
         (Ao-2 (send (eval plane) :point-into-equation
                                       (second (send self :endpoints))))
         (Ao (fourth (send (eval plane) :list-coeff))))
         (cond ((or (equal-error Ao Ao-1)
                    (equal-error Ao Ao-2))
            'nil)
           ((or (and (GE Ao-1 Ao)
                     (LE Ao-2 Ao))
                (and (LE Ao-1 Ao)
                     (GE Ao-2 Ao)))
             't))))


     (defmethod (line-segment :print) ()
       (pprint t-max *output-stream*)
       (pprint (list position-vector (send (eval position-vector) :list-format)
               (send (eval position-vector) :endpoints)) *output-stream*)
       (pprint (list direction-vector (send (eval direction-vector) :list-format)
               (send (eval direction-vector) :endpoints)) *output-stream*)
       (pprint (send self :endpoints) *output-stream*)
       (pprint characteristics *output-stream*))

     ;-------PLANE-------

     (defflavor plane                       ; uses equation of plane:
        (a-coef                     ;
         b-coef                     ; aX + bY + cZ = Ao
         c-coef                     ;
         Ao)                        ; for comparisions, equation is generally
         ()                         ; normalized, so Ao=-1,+1 or 0.
         :gettable-instance-variables       ; NOTE: first non-zero coeff will ALWAYS
be a
         :settable-instance-variables       ; positive number. Avoids direction
ambiguity
         :inittable-instance-variables
         :outside-accessible-instance-variables)

     (defmethod (plane :test-equal) (F2)    ; test plane for equality by comparing
                                            ; coefficients, or comparing the coeffs
                                            ; of the unit normal vectors
       (let ((V1 (init-vector '*origin* (init-point (send self :list-coeff-3))))
         (V2 (init-vector '*origin* (init-point (send (eval F2) :list-coeff-3)))))
         (or (apply 'and
             (map 'list #'equal-error
                      (send self :list-coeff)
                      (send (eval F2) :list-coeff)))
           (apply 'and
           (map 'list #'equal-error
                      (send (eval V1) :unit-vector)
                      (send (eval V2) :unit-vector)))))))
```

174

```
(defmethod (plane :list-coeff) ()   ; list plane coefficents as a 4-tuple
  (list a-coef b-coef c-coef Ao))    ; (includes the Ao constant term)

(defmethod (plane :list-coeff-3) ()   ; list only the x,y,z coefficents
  (list a-coef b-coef c-coef))

(defmethod (plane :subs-point-into-plane) (Pt)   ; subs a point into the planar
                                          ; equation, returns result.
  (apply '+ (map 'list '* (send self :list-coeff-3)
            (send (eval Pt) :list-format))))

(defmethod (plane :point-into-equation) (point)    ; subs point into plane
equation
                                          ; same as above
*****REMOVE****
  (apply '+ (map 'list '* (send (eval point) :list-format)
                  (send self : list-coeff-3))))

(defmethod (plane :find-x-given-yz) (y z)      ; find the x value of a point given
the
  (cond ((equal-zero-p a-coef) '0)      ; y and z coordinates of a point, for
                                          ; the plane under consideration
    (t (/ (- Ao (+ (* b-coef y) (*  c-coef z))) a-coef))))

(defmethod (plane :find-y-given-xz) (x z)      ; find the y value of a point given
the
  (cond ((equal-zero-p b-coef) '0)      ; x and z coordinates of a point, for
                                          ; the plane under consideration
    (t (/ (- Ao (+ (* a-coef x) (*  c-coef z))) b-coef))))

(defmethod (plane :find-z-given-xy) (x y)      ; find the z value of a point given
the
  (cond ((equal-zero-p c-coef) '0)      ; x and y coordinates of a point, for
                                          ; the plane under consideration
    (t (/ (- Ao (+ (* a-coef x) (*  b-coef y))) c-coef))))

(defmethod (plane :print) ()
  (pprint (send self :list-coeff) *output-stream*))


;---------BOUNDING BOX---------

(defflavor Bounding-box                  ; generalized bounding box flavor
    (max-x
     min-x
     max-y
     min-y
     max-z
     min-z)
    ()
    :gettable-instance-variables
    :settable-instance-variables
    :inittable-instance-variables
    :outside-accessible-instance-variables
    :required-methods)
```

```lisp
(defmethod (bounding-box :construct-bounding-box) (points)
                                         ; build bounding box for
                                         ; a list of points
  (let* ((first-point (send (eval (first points)) :list-format))
     (x (first first-point))
     (y (second first-point))
     (z (third first-point)))
    (setf max-x x)
    (setf min-x x)
    (setf max-y y)
    (setf min-y y)
    (setf max-z z)
    (setf min-z z)
    (loop for P in (rest points)
      do (let* ((next-point (send (eval P) :list-format))
            (new-x (first next-point))
            (new-y (second next-point))
            (new-z (third next-point)))
        (cond ((GT new-x max-x)
               (setf max-x new-x))
              ((LT new-x min-x)
               (setf min-x new-x)))
        (cond ((GT new-y max-y)
               (setf max-y new-y))
              ((LT new-y min-y)
               (setf min-y new-y)))
        (cond ((GT new-z max-z)
               (setf max-z new-z))
              ((LT new-z min-z)
               (setf min-z new-z))))))))

(defmethod (bounding-box :inside-bounding-box-p) (point)
                                      ; return T if point is inside
                                      ; bounding box, NIL otherwise
  (let ((p (map 'list '* (send (eval point) :list-format) '(1.0 1.0 1.0))))
    (cond ((and (and (GE max-x (first p))
                (LE min-x (first p)))
           (and (GE max-y (second p))
                (LE min-y (second p)))
           (and (GE max-z (third p))
                (LE min-z (third p))))
       't)
      (t 'nil))))


;-------FACET-------

(defflavor facet
   (edges                    ;list of all edges bounding facet
    center                   ; location of center of facet
    connects)                ; volumes which facet connects "((V1..Vn)
(V2..Vm))"
   (plane                    ;mixin flavors
    bounding-box)
   :gettable-instance-variables
   :settable-instance-variables
   :inittable-instance-variables
   :outside-accessible-instance-variables
   :required-methods)
```

```
(defmethod (facet :points) () ; return all verticies of facet
   (let ((temp 'nil))
     (loop for E in Edges
       do (setf temp (append temp (send (eval E) :endpoints))))
     (delete-duplicates temp)))

(defmethod (facet :find-facet-center) ()   ; find the average of all the vertices
                                           ; of the facet.
   (let* ((points (send self :points))
     (temp-sum (send (eval (first points)) :list-format))
     (nr-points (length points)))
     (loop for P in (rest points)
       do (setf temp-sum (map 'list '+ temp-sum
                                       (send (eval P) :list-format))))
     (setf (facet-center self)
       (init-point (map 'list '/ temp-sum (make-list 3 :initial-element
nr-points))))
     (facet-center self)))

(defmethod (facet :add-volume-to-left-connects) (V) ; add a volume to the left
list
                                          ; of the connects variable
   (cond ((null (facet-connects self))
     (setf (facet-connects self) (list (list V))))
     ((not (member-p V (first (facet-connects self))))
     (setf (first (facet-connects self)) (adjoin V (first (facet-connects
self)))))))))

(defmethod (facet :add-volume-to-right-connects) (V) ; add a volume to the right
list
                                          ; of the connects
variable
   (cond ((equal '1 (length (facet-connects self)))
     (setf (facet-connects self) (list (first (facet-connects self)) (list V))))
     ((not (member-p V (second (facet-connects self))))
     (setf (second (facet-connects self))
            (adjoin V (second (facet-connects self)))))))))


(defmethod (facet :print) ()
   (pprint (list edges center connects (send self :list-coeff)) *output-stream*))


;------VOLUME------

(defflavor volume
   (Visibility                        ; visible observers
    Probability-of-detection          ; sum of PD for observers
    Composition                       ; ground, air, etc
    Points                            ; all vertices of the volume
    Edges                             ; all line-segments of the volume
    Facets                            ; all surfaces of the volume
    Arithmetic-center                 ; numeric average of the points
    connected-to)                     ; adjacent volumes
   (Graphic)                          ; for 3-D projection
    :gettable-instance-variables
    :settable-instance-variables
    :inittable-instance-variables
    :outside-accessible-instance-variables
    :required-methods)
```

177

```lisp
(defmethod (volume :make-equal) (new-volume-name)
                                        ; make a new volume with same
instances
  (let ((temp new-volume-name))          ; as self
    (set temp (make-instance 'volume
                  :Visibility Visibility
                  :Probability-of-detection Probability-of-detection
                  :Composition Composition
                  :Points Points
                  :Edges Edges
                  :Facets Facets
                  :arithmetic-center Arithmetic-center
                  :connected-to Connected-to))))

(defmethod (volume :clear) ()    ; clear out old values of an existing volumes
  (setf Visibility 'nil)
  (setf Probability-of-detection 'nil)
  (setf Composition 'nil)
  (setf Points 'nil)
  (setf Edges 'nil)
  (setf Facets 'nil)
  (setf Arithmetic-center 'nil)
  (setf Connected-to 'nil))

(defmethod (volume :find-arithmetic-center) ()
                                        ; find the average of all the
vertices
                                        ; of the volume. do not change values
                                        ; in the volume
  (let ((temp-sum (send (eval (first Points)) :list-format))
    (nr-points (length Points)))
    (loop for P in (rest Points)
      do (setf temp-sum (map 'list '+ temp-sum
                             (send (eval P) :list-format))))
    (init-point (map 'list '/ temp-sum
                (make-list 3 :initial-element nr-points)))))

(defmethod (volume :make-node-list) ()
                                        ; make a list of absolute point coords in
graphic
  (loop for P in points               ; format (eg 4 element list)
                            ; used in GRAPHICS.
    collect (reverse (append (list '1)
                        (reverse (send (eval P) :list-format)))))))

(defmethod (volume :make-polygon-list) ()
                                        ;index point values to points in node
list
    (loop for L in edges               ; used in GRAPHICS
      do (setf Pt1 (car (send (eval L) :endpoint-list)))
      do (setf Pt2 (cadr (send (eval L) :endpoint-list)))
        collect (list (position-if '(lambda (A) (equal A Pt1)) node-list)
                  (position-if '(lambda (A) (equal A Pt2)) node-list))))

(defmethod (volume :print) ()
  (pprint (list Visibility Probability-of-detection Composition Points Edges
Facets
          arithmetic-center connected-to) *output-strerm*))
```

178

```
;------UNIVERSE-------

(defflavor Universe                          ; space of all volumes
    (Volumes
     Observers                               ; observers located within the defined
universe
     static-vectors                          ; save the state of the lines, points and
     static-vector-counter                   ; vectors used to build the static visibility
     static-lines                            ; model
     static-lines-counter
     static-points
     static-points-counter)
    ()
    :gettable-instance-variables
    :settable-instance-variables
    :inittable-instance-variables
    :outside-accessible-instance-variables)

(defmethod (universe :save-static-items) ()        ; save state of static
universe
    (setf static-vectors *list-of-vectors*)
    (setf *minimum-vectors-counter* *vectors-counter*)
    (setf static-lines *list-of-lines*)
    (setf *minimum-lines-counter* *lines-counter*)
    (setf static-points *list-of-points*)
    (setf *minimum-points-counter* *points-counter*)
    (setf *minimum-planes-counter* *planes-counter*)
    (setf *minimum-facets-counter* *facets-counter*)
    (setf *minimum-volumes-counter* *volumes-counter*))


;*******************************************************************************
;;;
;;; FUNCTIONS TO INITIALIZE; GET NAMES OF OBJECTS AND MAKE NAMES GLOBAL
;;;
;*******************************************************************************


(defun make-origin ()                                ; names of special points
and
    (gensym (incf *points-counter*))                 ; other unique flavors.
    (setf *origin* (make-instance 'point
                        :x-coord '0
                        :y-coord '0
                        :z-coord '0))
    (pushnew '*origin* *list-of-points*))



(defun make-null-vector ()
    (gensym (incf *vectors-counter*))
    (setf *null-vector* (make-instance 'vector
                        :i '0
                        :j '0
                        :k '0
                        :Start-point '*origin*
                        :End-point '*origin*))
    (push '*null-vector* *list-of-vectors*))
```

179

```lisp
(defun make-point-name ()                    ;produce variable names "on the fly"
  (cond ((> *points-counter* (1- *max-counter-value*))
     (setf *points-counter* *minimum-points-counter*)))
  (gensym (incf *points-counter*))
  (intern (gensym "point")))

(defun make-line-name ()
  (cond ((> *lines-counter* (1- *max-counter-value*))
     (setf *lines-counter* *minimum-lines-counter*)))
  (gensym (incf *lines-counter*))
  (intern (gensym "line")))

(defun make-vector-name ()
  (cond ((> *vectors-counter* (1- *max-counter-value*))
     (setf *vectors-counter* *minimum-vectors-counter*)))
  (gensym (incf *vectors-counter*))
  (intern (gensym "vector")))

(defun make-facet-name ()
  (cond ((> *facets-counter* (1- *max-counter-value*))
     (setf *facets-counter* *minimum-facets-counter*)))
  (gensym (incf *facets-counter*))
  (intern (gensym "facet")))

(defun make-plane-name ()
  (cond ((> *planes-counter* (1- *max-counter-value*))
     (setf *planes-counter* *minimum-planes-counter*)))
  (gensym (incf *planes-counter*))
  (intern (gensym "plane")))

(defun make-volume-name ()
  (cond ((> *volumes-counter* (1- *max-counter-value*))
     (setf *volumes-counter* *minimum-volumes-counter*)))
  (gensym (incf *volumes-counter*))
  (intern (gensym "volume")))

;********************************************************************
;;;
;;;   FLAVOR INSTANTIATION FUNCTIONS
;;;
;;;   Note: all of therse functions will stop keeping lists of previously
;;;         created instantiations after flag
;;;         *done-making-new-visibility-volumes-flag* is set to T
;********************************************************************


;-------MAKE A POINT--------

(defun init-point (List-of-values)    ; see if point already exists
(nonrecursive)
  (cond ((and (not (null *list-of-points*))
          (no' done-making-new-visibility-volumes-flag*))
     (loop fc. P in *list-of-points*
       do (cond ((apply 'and
                   (map 'list #'equal-error
                        (map 'list 'rationalize list-of-values)
                        (send (eval P) :list-format))'
                 (return-from init-point P))))))
    (init-new-point list-of-values))
```

180

```lisp
(defun init-new-point (List-of-values)
  (let ((temp (make-point-name)))
    (set temp (make-instance 'point
                     :x-coord (rationalize (first List-of-values))
                     :y-coord (rationalize (second List-of-values))
                     :z-coord (rationalize (third List-of-values))))
    (push temp *list-of-points*)
    temp))

;--------MAKE A VECTOR-------

(defun init-vector (Start-point End-point)    ; check to see if vector already
built
  (cond ((not *done-making-new-visibility-volumes-flag*)
     (loop for V in *list-of-vectors*
            do (cond ((equal (send (eval V) :endpoints)
                  (list Start-point End-point))
                    (return-from init-vector V))))))
  (init-new-vector Start-point End-point))

(defun init-new-vector (Sp Ep)
  (let ((temp (make-vector-name)))
    (set temp
      (make-instance 'vector
                  :i (- (point-x-coord (eval Ep)) (point-x-coord (eval Sp)))
                  :j (- (point-y-coord (eval Ep)) (point-y-coord (eval Sp)))
                  :k (- (point-z-coord (eval Ep)) (point-z-coord (eval Sp)))
                  :Start-point Sp
                  :End-point Ep))
    (push temp *list-of-vectors*)
    temp))




;--------MAKE A LINE SEGMENT-------

(defun init-line (Position-vector Direction-vector)
                                       ; valid construction for a line???
    (cond ((and (equal (vector-Start-point (eval Position-vector)) '*origin*)
                  (member-p (vector-end-point (eval Position-vector))
                                    (send (eval Direction-vector) :endpoints)))
            (Find-or-make-line  Position-vector Direction-vector))
            (t (terpri)
               (princ "Error invalid vectors: ")
               (prin1 (list position-vector direction-vector)) (terpri))))

(defun Find-or-make-line (Pv Dv)              ; check to see if line already
built
  (cond ((not *done-making-new-visibility-volumes-flag*)
     (loop for L in *list-of-lines*
            do (cond ((and (member-p (vector-end-point (eval Pv))
                          (send (eval (old-line-Dv L)) :endpoints))
                    (or (equal (send (eval Dv) :endpoints)
                      (send (eval (old-line-Dv L)) :endpoints))
                        (equal (send (eval Dv) :endpoints)
                           (nreverse
                            (send (eval (old-line-Dv L)) :endpoints)))))
                  (return-from find-or-make-line L))))))
  (init-new-line Pv Dv))
```

```lisp
(defun init-new-line (Pv Dv)
  (let ((temp (make-line-name)))
    (set temp (make-instance 'line-segment
                      :t-max '1
                      :Position-vector Pv
                      :Direction-vector Dv
                      :characteristics 'nil))
    (push temp *list-of-lines*)
    temp))

(defun old-line-Dv (Line)
  (line-segment-Direction-vector (eval Line)))


;----------MAKE A PLANE---------


(defun init-plane (List-of-values)    ; see if plane already exists
(nonrecursive)
  (cond ((and (not (null *list-of-planes*))
          (not *done-making-new-visibility-volumes-flag*))
     (loop for P in *list-of-planes*
        do (cond ((or (equal (send (eval P) :list-coeff)
                           list-of-values)
                     (apply 'and (map 'list #'equal-error
                                        (send (eval P) :list-coeff)
                                         list-of-values)))
                  (return-from init-plane P))))))
    (init-new-plane list-of-values))

(defun init-new-plane (List-of-values)
  (let ((temp (make-plane-name)))
    (set temp (make-instance 'plane
                      :a-coef (rationalize (first list-of-values))
                      :b-coef (rationalize (second list-of-values))
                      :c-coef (rationalize (third list-of-values))
                      :Ao (fourth list-of-values)))
    (push temp *list-of-planes*)
    temp))


;-------------------------MAKE ALL FACETS----------------------------
;
; Used by intercept routines to rebuild volume facets.
;
;
;                   ***   W A R N I N G   ***
;
; Note: Facets MUST be convex and MUST NOT be adjacent to
;       facets in the same volume with the same plane equation
;
;--------------------------------------------------------------------
; Used by input method 1 and by all intercept routines

(defun make-all-facets (Volume)
  (reset-point-property-lists Volume)
                                    ; initialize point 'lines property list
  (loop for L in (Volume-edges (eval Volume))
    do (let* ((endpoints (send (eval L) :endpoints))
          (first-point (first endpoints))
          (second-point (second endpoints)))
      (setf (get first-point 'lines) (adjoin L (get first-point 'lines)))
```

182

```lisp
                (setf (get second-point 'lines) (adjoin L (get second-point 'lines)))))))
                                            ; build all facets from points
         (loop for P in (volume-points (eval Volume)) ; make all facets possible
            do (loop for L in (get P 'lines)
            do (let* ((other-end-L (send (eval L) :other-end P)))
                   (initialize-search Volume P (list L) (List other-end-L P))))))

    (reset-point-property-lists Volume))

  (defun initialize-search (Volume  Goal old-lines old-points)
    (let ((point2 (first old-points))
      (Line (first old-lines))
      (search-result 'nil)
      (facet-name 'nil))
      (loop for L in (get point2 'lines)
        do (cond ((and (not (equal L Line))
                    (not (equal Goal ('end (eval L) :other-end point2))))
              (let ((plane (init-plane (make-a-normalized-plane L Line))))
                (cond ((not (member-p plane (get Goal 'planes)))
                    (setf (get Goal 'planes) (adjoin plane (get Goal 'planes)))
                    (setf search-result (search-to-make-facet Goal
                                        plane
                                        (list L Line)
                                        (pushnew (send (eval L) :other-end point2)
                                                old-points)
                                        'nil
                                        'nil))
                    (cond ((<= '3 (length (first search-result)))
                        (setf facet-name (init-facet-2 search-result)))
                        (t (setf facet-name 'nil)))
                    (cond ((not (null facet-name))
                            (setf (volume-facets (eval Volume))
                                (adjoin facet-name (volume-facets (eval
Volume)))))
                    ))))))))))

  (defun search-to-make-facet (Goal                   ;
                    Facet-plane         ;
                    old-lines           ;
                    old-points          ;
                    rejected-points   ;
                    rejected-lines)    ;
    (let ((current-point (first old-points))
      (last-line (first old-lines))
      (Line 'nil)
      (possible-paths 'nil))
        (loop for candidate-line in (get current-point 'lines)
            do (let ((other-end-cand-line
                (send (eval candidate-line) :other-end current-point)))
                (cond ((apply 'and (list (not (member-p candidate-line old-lines))
                                (not (member-p candidate-line rejected-lines))
                                (not (member-p other-end-cand-line
                                            rejected-points))))
                    (cond ((not (member-p other-end-cand-line old-points))
                        (cond ((send (eval facet-plane) :test-equal
                                (make-a-plane other-end-cand-line
                                            (first old-lires)))
                            (setf (get other-end-cand-line 'distance)
                                (distance Goal other-end-cand-line))
                            (setf possible-paths
                                (adjoin candidate-line possible-paths)))
                            (t (pushnew candidate-line rejected-lines))))))
```

183

```
                  ((equal other-end-cand-line Goal)
                   (loop for P in (adjoin other-end-cand-line old-points)
                     do (setf (get P 'planes)
                              (adjoin Facet-plane (get P 'planes))))
                   (return-from search-to-make-facet (list
                                                (adjoin candidate-line
                                                        old-lines)
                                                facet-plane)))
                  (t (pushnew candidate-line rejected-lines))))
              (t (pushnew candidate-line rejected-lines)))))
      (cond ((not (null possible-paths))
             (setf Line (minimum-distance possible-paths current-point))
        (push Line old-lines)
        (pushnew (send (eval Line) :other-end current-point) old-points))
       (t (pushnew last-line rejected-lines)       ; remove last line, current
point
          (pushnew current-point rejected-points) ; and retrace steps (backtrack)
          (setf old-lines (rest old-lines))
          (setf old-points (rest old-points))
          (cond ((> 2 (length old-lines))       ; backtracked too far?
              (return-from search-to-make-facet 'nil)))))
      (search-to-make-facet Goal Facet-plane old-lines old-points
                                  rejected-points rejected-lines)))


(defun init-facet-2 (properties)        ; Check to see if already built facet
  (cond ((not (null properties))        ; else return name of new facet, or nil.
         (let* ((edges (first properties))
                (plane (second properties))
                (test-plane (map 'list 'abs
                         (map 'list '* (send (eval plane) :list-coeff)
                                  *one-vector*)))
                (equal-flag 't))
          (cond ((equal-p test-plane *zero-vector*) ; remove artifact facets
              (return-from init-facet-2 'nil)))
              (cond ((not (null *list-of-facets*))
                 (loop for F in *list-of-facets*     ; see if already exists
                   do (cond ((equal (length edges)
                             (length (facet-edges (eval F))))
                        (setf equal-flag 't)
                        (loop for E in edges
                        do (cond ((not (member-p E (face -edges (eval F))))
                                  (setf equal-flag 'nil))))
                        (cond (equal-flag
                              (return-from init-facet-2 F)))))))
            (make-new-facet edges plane)))
    (t (return-from init-facet-2 'nil))))


(defun make-new-facet (list-of-edges plane)
  (let ((plane-equation (send (eval Plane) :list-coeff))
    (temp (make-facet-name)))
    (set temp (make-instance 'facet
                  :Edges list-of-edges
                  :center 'nil
                  :connects 'nil
                  :a-coef (first Plane-equation)
                  :b-coef (second Plane-equation)
                  :c-coef (third Plane-equation)
                  :Ao (fourth Plane-equation)))
    (push temp *list-of-facets*)
    temp))
```

184

```
;----------------------MAKE A FACET FROM INPUT-----------------------
; Used by input method 2 (only)


(defun make-a-facet (points)      ; build a facet from a list of point names
   (let ((first-point (first points))
     (start-point (first points))
     (lines 'nil)
     (plane-of-facet 'nil))
     (loop for End-point in (rest points)    ; construct edges of facet
       do (let ()
         (setf lines (adjoin (make-line Start-point End-point) lines))
         (setf Start-point End-point)))
       (setf lines (adjoin (make-line Start-point First-point) lines))

       (setf Plane-of-facet (init-plane (make-a-normalized-plane (first lines)
                                                 (second lines))))
       (make-new-facet lines plane-of-facet)))   ; return new facet name


(defun build-side-facet (Pt1 Pt2 Side-points) ; make a facet w/disjoint list of
points
   (make-a-facet (append (list Pt1 Pt2) Side-points)))

(defun build-terrain (data)       ; build facets with raw facet data, where data
                                  ; is in format (point point point ...)
                                  ; and the points are in format (x y z)
                                  ; return a list of all facets built
   (let ((list-of-facets 'nil))
     (loop for Facets in Data      ; each list within data is a facet
       do (let ((points (map 'list #'init-point Facets)))
       (setf list-of-facets (adjoin (make-a-facet points) list-of-facets))))
   list-of-facets))


;---------------------------MAKE A VOLUME---------------------------

(defun init-volume (data)
   (let ((temp (make-volume-name))
     (volume-data (pop data)))
     (set temp (make-instance 'volume
                       :Visibility (second volume-data)
                       :Probability-of-detection 'nil
                       :Composition (first volume-data)
                       :Points '()
                       :Edges '()
                       :Facets '()
                       :arithmetic-center 'nil
                       :connected-to 'nil))
     (push temp *list-of-volumes*)
     (Initialize-volume temp data)
     temp))                                   ; return name of volume created
```

185

```
;*********************************************************************
;
;         C O N S T R U C T I O N    U T I L I T Y    F U N C T I O N S
;
;    -----------------------------------------------------------------
;
;*********************************************************************


(defun sample-2-1 ()
  (set-up 1 '"t27-ridges-shadow")
  (init-observer '(500 50 200) '0.02)
  (set-up-2)
  (pprint (length (universe-volumes *universe*)))
  (a-star-search (init-point '(0 0 200)) (init-point '(0 1000 200)) 'nil 'nil))

(defun sample-4-2 ()
  (set-up 2 '"t310-full-ridge")
  (init-observer '(10 500 250) '0.75)
  (init-observer '(990 500 250) '0.50)
  (set-up-2)
  (pprint (length (universe-volumes *universe*)))
  (a-star-search (init-point '(500 10 400)) (init-point '(500 990 400)) 'nil
'nil))

(defun sample-5-1 ()                  ; one obs in central valley
  (set-up 1 '"t25-ridge-box")
  (init-observer '(0 500 200) '0.02)
  (set-up-2)
  (pprint (length (universe-volumes *universe*)))
  (a-star-search (init-point '(10 10 500)) (init-point '(10 990 225)) 'nil
'nil))

(defun sample-5-2 ()                  ; one obs in central valley, one on side
  (set-up 1 '"t25-ridge-box")
  (init-observer '(0 500 200) '0.75)
  (init-observer '(50 50 250) '0.75)
  (set-up-2)
  (pprint (length (universe-volumes *universe*)))
  (a-star-search-m (init-point '(10 10 500)) (init-point '(10 990 225)) 'nil '10
'nil))

(defun sample-6-1 ()                  ; single observer on one side of central
valley
  (set-up 2 '"t320-double-peak")
  (init-observer '(10 500 225) '0.02)
  (set-up-2)
  (pprint (length (universe-volumes *universe*)))
  (a-star-search (init-point '(500 10 250)) (init-point '(500 990 250)) 'nil
'nil))

(defun sample-6-2 ()                  ; one each on each side of the peaks
  (set-up 2 '"t320-double-peak")
  (init-observer '(10 250 250) '0.75)
  (init-observer '(990 750 250) '0.75)
  (set-up-2)
  (pprint (length (universe-volumes *universe*)))
  (a-star-search (init-point '(500 10 250)) (init-point '(500 990 250)) 'nil
'nil))
```

186

```
(defun sample-7-1 ()
  (set-up 2 '"t360-2-peak-2-ridge")
  (init-observer '(100 800 250) '0.75)
  (set-up-2)
  (pprint (length (universe-volumes *universe*)))
  (a-star-search (init-point '(10 10 300)) (init-point '(990 990 300)) 'nil
'nil))

(defun sample-8-1 ()
  (set-up 2 '"t350-six-peaks")
  (init-observer '(100 800 250) '0.75)
  (set-up-2)
  (pprint (length (universe-volumes *universe*)))
  (a-star-search (init-point '(10 10 300)) (init-point '(500 990 300)) 'nil
'nil))

(defun sample-final-paths ()
  (let ((goal (init-point '(10 990 225)))
    (list-of-start-points 'nil))
    (loop for C from 10 upto 990 by 100
      do (setf list-of-start-points (adjoin (init-point (list C '10 '600))
                                      list-of-start-points)))
    (loop for S in (reverse list-of-start-points)
      do (speed-demon)
      do (a-star-search S Goal 'nil'nil))
    (display-paths *list-of-paths*)))

(defun sample-9-1 ()                    ; one obs in central valley
  (set-up 1 '"t21-ridge-Y")
  (init-observer '(990 500 200) '0.0150)
  (set-up-2)
  (pprint (length (universe-volumes *universe*)))
  (a-star-search (init-point '(10 10 410)) (init-point '(10 990 410)) 'nil
'nil))
```

```
;;; -*- Mode:Common-Lisp; Base:10 -*-

(defvar testvar)

(defun TS ()
  (a-star-search (init-point '(0 0 200)) (init-point '(0 1000 200)) 'nil 't))


(defun TS1 ()
  (a-star-search (init-point '(0 0 200)) (init-point '(0 1000 200)) 't 'nil))


(defun TS2 ()
  (a-star-search-M (init-point '(0 0 200)) (init-point '(0 1000 200)) 't 5
'nil))


(defun TS3 ()
  (a-star-search-M (init-point '(0 0 200)) (init-point '(0 1000 200)) 'nil 10
'nil))

(defun TS4 ()
  (a-star-search-M (init-point '(0 0 200)) (init-point '(0 1000 200)) 'nil 5
't))

(defun TS5 () ;used with box-canyon or t-27-ridge-shadow
  (a-star-search-M (init-point '(510 0 800)) (init-point '(500 1000 900)) 'nil 5
't))

(defun TS6 () ;used with box-canyon
  (setf testvar (a-star-search-M (init-point '(900 0 300)) (init-point '(990
1000 250)) 'nil 5 't)))

(defun TS7 () ;used with or t-27-ridge-shadow
  (setf testvar(a-star-search-M (init-point '(0 310 210)) (init-point '(1000 750
300)) 'nil 5 't)))

(defun TS8 () ;used with or t-27-ridge-shadow
  (a-star-search-M (init-point '(10 0 300)) (init-point '(990 990 990)) 'nil 5
't))

(defun TS9 () ;used with or t-27-ridge-shadow
  (a-star-search-M (init-point '(900 10 910)) (init-point '(1000 750 300)) 'nil
5 't))

(defun TS340-1 ()
  (a-star-search (init-point '(950 0 210)) (init-point '(990 1000 550)) 'nil
'nil))

(defun ts-speed (path-list)
  (let ((time1)
    (time2))
    (setf time1 (time))
    (random-ray-optimize path-list)
    (setf time2 (time))
    (princ "this is time1 - ")(princ time1)(terpri)
    (princ "this is time2 - ")(princ time2)(terpri)
    (princ "the difference is - ")(princ (- time2 time1))(terpri)))


(defun ts-speed-old-opt (path number-of-optimizations)
  (let ((time1)
```

188

```lisp
         (time2))
          (setf time1 (time))
          (do* ((new-path (optimize-path path) (optimize-path new-path))
            (count (- number-of-optimizations 1)(- count 1)))
           ((zerop count) (princ path)(path-data new-path)))
          (setf time2 (time))
          (princ "this is time1 - ")(princ time1)(terpri)
          (princ "this is time2 - ")(princ time2)(terpri)
          (princ "the difference is - ")(princ (- time2 time1))(terpri)))


(defun TS25-1 ()
   (setf testvar (a-star-search-m (init-point '(950 0 510))
                          (init-point '(990 1000 550))
                          'nil
                          5
                          't))
   )


(defun TS10 ()
   (setf testvar
    (a-star-search-m (init-point '(10 400 910)) (init-point '(110 990 450)) 'nil
5 'nil)))

(defun TS11 () ;used with or t-27-ridge-shadow
   (setf testvar(a-star-search-M (init-point '(0 0 990)) (init-point '(1000 750
300)) 'nil 5 't)))

(defun TS12 () ;used with t-27 for user adjustment
   (a-star-search-M (init-point '(410 10 900)) (init-point '(900 990 300)) 'nil 5
't))

(defun user-adj (point)
   (let* ((P1 (init-point point))
      (line (make-line (init-point '(410 10 900)) P1))
      (path (revise-path '|path0006| line)))
      (path-data '|path0006|)
      (princ path)
      (path-data path)
      (path-for-iris path)))

(defun TS13 () ;used with t-27 for user adjustment
   (a-star-search-M (init-point '(300 10 450)) (init-point '(990 900 250)) 'nil 5
't))


(defun TS14 ()
   (a-star-search (init-point '(500 200 600)) (init-point '(300 990 440)) 'nil
'nil))
```

189

APPENDIX B

This Appendix contains a listing of the following files:
>ppgh.c
>rotate.h
>lightdef.c
>lightdef.h
>Makefile

Instructions for use of graphics program:

1. Load all the above files along with base1.dat, ground1.dat, and path1.dat in
    one directory on the graphic workstations.

2. Type *make<cr>*.

3. Type *ppgh<cr>*.

4. Open the window with the mouse and observe the paths and terrain.

5. To exit click on the right mouse button and select "exit".

```
/* this is an IRIS-4D Program */

/* this is file ppgh.c  short for path planning graphics
   This program is used in conjunction with data files created
   by a LISP Optimal Path Planning Program

   It is  an alteration  of program rotate2.c with z-buffering and rgbmode
   with polygon removal and lighting routines...

*/


#include "gl.h"
#include "device.h"
#include "math.h"
#include "rotate.h"
#include "stdio.h"
#include "lightdefs.h"    /* get the material/light/lighting model defs */

#define NEARDEPTH 0x00000  /* presently set for the 4D/GT */
#define FARDEPTH  0x7ffff

#define NEARCLIPPING 10.0   /* near clipping plane is at -10.0 */
#define FARCLIPPING  6000.0 /* far clipping plane is at -6000.0 */

#define CUBEX     500.0    /* location of the cube */
#define CUBEY     300.0    /* actually the center  */
#define CUBEZ     -500.0   /* of our terrain model */
#define CUBESIZE 100.0

#define VIEWX     500.0    /* initial location of the viewpoint */
#define VIEWY     500.0
#define VIEWZ     2000.0

#define REFX      CUBEX    /* reference point we are looking towards */
#define REFY      CUBEY
#define REFZ      CUBEZ

#define PI 3.1416
#define MAXPOINTS 5

float viewx = 500.0;   /* location of the viewpoint */
float viewy = 300.0;
float viewz = 1000.0;
float val1;
float viewingdistance = 1500.0;   /* initial distance from center of obj */
float viewingangle = 0.0;     /* angle in YZ plane at which obj is viewed */

int radius,val0;
int sens = 10;

typedef struct threedobj {
    int numpts;
    float point[MAXPOINTS][3];
    float normal[3]
    };
struct threedobj base[20], ground[100];
```

191

```
int numbasepts, numgroundpts;

int numberofpaths;   /* this is the number of paths in the file max is 10 */
int numberofwaypoints[10];   /* this is the max num of waypoints for the
fltpath*/
float waypoints[10][100][4]; /*array for storing the flt path*/

   long xwinsize, ywinsize ;


main()
{

   /* popup menu's name */
   int mainmenu;

   int threed, twod, help;   /* window numbers */

   int hititem; /* variable holding hit name */

   short val, valsave; /* value returned from the event queue */
    char filename[20];
    int i,j;


   /* initialize the IRIS system */
   initialize(&threed, &twod);

   /* initialize the material definitions */
   initializematerials();

   /* initialize the light definitions */
   initializelights();

   /* initialize the lighting model */
   initializelmodel();

   /* reset dials */
   setdial0();
   setdial1();
   setdial2();

   /* make the popup menus */
   mainmenu = makethemenus();

   /* set all the accumulative matrices to unit matrices */
   resetallaccumulativematrices();

   /* go get the polygons that make up the base */
    strcpy(filename, "base1.dat");
    inputstructure(base, &numbasepts, filename);


   /* compute the normals for the base */
    calculatenormal(base,numbasepts, 500.0, 0.0, -500.0);
```

```c
#ifdef TRACE
    /* print out the input */
    for (i=0; i<numbasepts; i++)
    {
        printf("number of points =%d\n",base[i].numpts);
        printf("normal for %d is %lf %lf %lf \n",i,base[i].normal[0],
            base[i].normal[1],
            base[i].normal[2]);


    for (j=0; j<base[i].numpts; j++)
        {
        printf("points[%d][0]=%.f  points[%d][1]=%.f  points[%d][2]=%.f\n",
                j,base[i].point[j][0],
                j,base[i].point[j][1],
                j,base[i].point[j][2]);
        }
    }
#endif
    /* input the ground data */
    strcpy(filename, "ground1.dat");
    inputstructure(ground, &numgroundpts, filename);
    calculatenormal(ground,numgroundpts, 500.0, -5000.0, -500.0);

#ifdef TRACE
    for (i=0; i<numgroundpts; i++)
    {
        printf("number of points =%d\n",ground[i].numpts);
        printf("normal for %d is %lf %lf %lf \n",i,
            ground[i].normal[0],
            ground[i].normal[1],
            ground[i].normal[2]);


    for (j=0; j<ground[i].numpts; j++)
        {
        printf("points[%d][0]=%.f  points[%d][1]=%.f  points[%d][2]=%.f\n",
                j,ground[i].point[j][0],
                j,ground[i].point[j][1],
                j,ground[i].point[j][2]);
        }
    }
#endif

/* get the input for the flight path */
    inputlinearray();
#ifdef TRACE
    printf("this is the set input from the file\n");
    for(i=0; i<numberofwaypoints; i++)
    {

        printf("i= %d  x= %f  y= %f  z= %f percentdetection= %f\n",
                i,waypoints[i][0],waypoints[i][1],
                waypoints[i][2], waypoints[i][3]);
    }
#endif
```

```
while(TRUE)
{

    /* do we have something on the event queue ?*/
    if(qtest())
    {

        switch(qread(&val))
        {

            case MENUBUTTON:

                if(val == 1)
                {
                    /* we must be in MSINGLE mode to do popup menus!!! */
                    mmode(MSINGLE);

                    /* which popup selection do we want? */
                    hititem = dopup(mainmenu);

                    /* put us back into MVIEWING mode */
                    mmode(MVIEWING);

                    /* do something with the popup hit */
                    processmenuhit(hititem);

                }

                break;

    case DIAL0:


            val0 = (int)((val / sens));
            if (valsave < val)
                    ry = val0 * YROTAMOUNT;

            if (valsave > val)
                    ry = val0 * YROTAMOUNT;



            buildmovingviewingmatrix(viewx,viewy,viewz,REFX,REFY,REFZ);

            ry = 0;
            setdial0();
/*          valsave = val;*/
            break;

        case DIAL1:

            viewingangle = val/sens;
            viewy = (float)sin((double)(viewingangle * PI / 180.0))*
             viewingdistance + REFY;
            viewz = (float)cos((double)(viewingangle * PI / 180.0))*
             viewingdistance + REFZ;

            break;
```

194

```
        case DIAL2:


           viewingdistance = (float)val * 10;
           viewy = (float)sin((double)(viewingangle * PI / 180.0))*
            viewingdistance + REFY;
           viewz = (float)cos((double)(viewingangle * PI / 180.0))*
            viewingdistance + REFZ;
#ifdef TRACE
 printf("val is %d    viewingdist is %f   viewy is %f  viewz is %f\n\n",
       val,viewingdistance,viewy,viewz);
#endif

           break;


          case REDRAW:

             reshapeviewport();

             break;

          default:

             break;

       }   /* end switch on event queue item */

     }   /* endif qtest()   */


     /* draw the background color */
     RGBcolor(150,150,150); /* grey */
     clear();

     /* turn on Z-buffering */
     zbuffer(TRUE);

     /* clear the z-buffer */
     zclear();

     /* put up the non-moving viewing matrix for the meter */
     buildnonmovingviewingmatrix(VIEWX,VIEWY,VIEWZ,REFX,REFY,REFZ);


     /* display the number of frames per second
        (0.0,40.0,-20.0) is the loc of the meter in world coords.
        15.0 is the radius to use for the meter.
        40.0 is maximum frames per second.
        Note: this measures actual CPU time used by your
              process and other users on the system will
              make the output meter view strange.
     */
     zbuffer(FALSE);
     lmbind(LMODEL,0);   /* turn off lighting model */

      zbuffer(TRUE);
     lmbind(LMODEL,MYMODEL);   /* turn my lighting back on */
```

```c
        /* put up the moving viewing matrix.
           The input arguments are the center point for the object
           we want to move and the reference point in the scene.
           We need this guy so we can always rotate around
           the screen frame of reference.
        */
        buildmovingviewingmatrix(viewx,viewy,viewz,REFX,REFY,REFZ);

        /* draw the base */
        lmbind(MATERIAL, DIRT);
        drawobject(base,numbasepts);

        /* draw the ground */
        lmbind(MATERIAL, GRASS2);
        drawobject(ground,numgroundpts);

        /*draw path */
        drawpath();

        /* turn z-buffering off */
        zbuffer(FALSE);


        /* change the buffers ... */
        swapbuffers();

    }

}

initialize(threed, twod)
int *threed, *twod; /* for display window and controles window */
{
int xorigin, yorigin;
    /* set up the preferred aspect ratio */
    keepaspect(XMAXSCREEN+1,YMAXSCREEN+1);

    /* open main window  */
    winopen("main");
    /* get the size of main */
    getsize(&xwinsize, &ywinsize);
    printf("x= %ld y= %ld \n",xwinsize, ywinsize);

    getorigin(&xorigin, &yorigin);
    printf("x= %ld y= %ld \n",xorigin, yorigin);

    /* set the size of the path window */
    prefposition(xorigin, xorigin + xwinsize,
            yorigin, yorigin + ywinsize);


    /* open a window for the program */
    *threed = winopen("path");

    /* make a title */
    wintitle("Path Planning");

    /* put the IRIS into double buffer mode */
    doublebuffer();

    /* put the iris into rgb mode */
    RGBmode();
```

196

```
    /* configure the IRIS (means use the above command settings) */
    gconfig();
    /* set the depth for z-buffering only for GT*/
    lsetdepth(NEARDEPTH,FARDEPTH);


    /* queue the redraw device */
    qdevice(REDRAW);

    /* queue the menubutton */
    qdevice(MENUBUTTON);

    /* queue up dials */
    qdevice(DIAL0);
    qdevice(DIAL1);
    qdevice(DIAL2);

    /* turn the cursor on */
    curson();

    /* select gouraud shading */
    /* only works on the 4D*/
    shademodel(GOURAUD);

    /* turn on the new projection matrix mode */
    mmode(MVIEWING);

}

/* this routine resets/initalized the dial 0   */
setdial0()
{
  setvaluator(DIAL0,
          (0*sens),
          (-360 * sens),
          ( 360 * sens));
  qreset();
}
/* this routine resets/initalized the dial 1   */
setdial1()
{
  setvaluator(DIAL1,
          (0*sens),
          ( 0 * sens),
          ( 89 * sens));
  qreset();
}

/* this routine resets/initalized the dial 2   */
setdial2()
{
  setvaluator(DIAL2,
          (int)(viewingdistance/sens),
          (0),
          (500));
  qreset();
}
```

```c
/* this routine performs all the menu construction calls */

int makethemenus()
{
    int topmenu;    /* top level menu's name */
    int rotmenu;    /* rotate menu */
    int transmenu;  /* trans menu   */
    int scalemenu;  /* scale menu   */

    /* define the low level menus first */
    rotmenu = newpup();
    addtopup(rotmenu,"Rotate Menu %t ");
    addtopup(rotmenu,"+X %x2 | OX %x3 | -X %x4 ");
    addtopup(rotmenu,"+Y %x5 | OY %x6 | -Y %x7 ");
    addtopup(rotmenu,"+Z %x8 | OZ %x9 | -Z %x10 ");

    transmenu = newpup();
    addtopup(transmenu,"Translate Menu %t ");
    addtopup(transmenu,"+X %x12 | OX %x13 | -X %x14 ");
    addtopup(transmenu,"+Y %x15 | OY %x16 | -Y %x17 ");
    addtopup(transmenu,"+Z %x18 | OZ %x19 | -Z %x20 ");

    scalemenu = newpup();
    addtopup(scalemenu,"Scale Menu %t ");
    addtopup(scalemenu,"+X %x22 | OX %x23 | -X %x24 ");
    addtopup(scalemenu,"+Y %x25 | OY %x26 | -Y %x27 ");
    addtopup(scalemenu,"+Z %x28 | OZ %x29 | -Z %x30 ");

    /* build the top level menu */
    topmenu = defpup("Roll Off Side %t | Rotate %x1 %m | Translate %x11 %m |
Scale %x21 %m | Reset %x31 | Exit %x32 ",
                     rotmenu,transmenu,scalemenu);

    /* return the name of this menu */
    return(topmenu);

}
```

```
/* this routine builds the moving viewing matrix each time through
   the display loop...

   P' = P . T(to origin) . S(acc) . R(x acc) . R(y acc) . R(z acc)
             . T(to acc. loc) . T(back to specified center) . perspective() */

buildmovingviewingmatrix(vx,vy,vz,refx,refy,refz)

float vx,vy,vz;    /* view point */
float refx,refy,refz;    /* ref point we are looking towards */
{
   /* Build the accumulative rotation matrices */
   loadmatrix(rxacc); /* get the accumulative rotation */
   rotate(rx,'x');    /* concatenate on the new rotation (if any) */
   getmatrix(rxacc);  /* we now have a new accumulative... */

   loadmatrix(ryacc); /* get the accumulative rotation */
   rotate(ry,'y');    /* concatenate on the new rotation (if any) */
   getmatrix(ryacc);  /* we now have a new accumulative... */

   loadmatrix(rzacc); /* get the accumulative rotation */
   rotate(rz,'z');    /* concatenate on the new rotation (if any) */
   getmatrix(rzacc);  /* we now have a new accumulative... */

   /* Build the accumulative translation matrix */

   loadmatrix(transacc); /* get the accumulative translation   */
   translate(tx,ty,tz);  /* concatenate on the new translation */
   getmatrix(transacc);  /* we now have a new accumulative translation */

   /* Build the accumulative scale matrix */
   loadmatrix(scaleacc); /* get the accumulative scale */
   scale(sx,sy,sz);      /* concatenate on the new scale */
   getmatrix(scaleacc);  /* we now have the new accumulative scale */

   /* in mmode(MVIEWING), we must add a load of a unit matrix */
   loadunit();

   /* put up the projection and viewing matrix */
   projectionandviewingmatrix(vx,vy,vz,refx,refy,refz);

   /* translate center of box back to original location */
   translate(refx,refy,refz);

   /* translate the object to the location specified
      by the accumulative translation...
   */
   multmatrix(transacc);

   multmatrix(rzacc);   /* z accumulative matrix */

   multmatrix(ryacc);   /* y accumulative matrix */

   multmatrix(rxacc);   /* x accumulative matrix */

   multmatrix(scaleacc); /* accumulative scale matrix */

   /* translate center of box to the origin */
   translate(-refx,-refy,-refz);
}
```

```c
/* for objects that are in the same coordinate system but aren't moving
   with the continuous rotations/translations/scalings, we use this
   routine ...  */

buildnonmovingviewingmatrix(vx,vy,vz,refx,refy,refz)

float vx,vy,vz;    /* view point */
float refx,refy,refz; /* reference point we are looking towards    */
{

   /* we must call loadunit before we get the projection
      and viewing stuff... */
   loadunit();

   /* just call the perspective + viewing matrices */
   projectionandviewingmatrix(vx,vy,vz,refx,refy,refz);

}


/* put up the projection and viewing matrix */

projectionandviewingmatrix(vx,vy,vz,refx,refy,refz)

float vx,vy,vz;    /* view point */
float refx,refy,refz;    /* reference point */
{

   /* perspective projection 3D for the world coord sys */
   /* the near and far values are distances from the viewer
      to the near and far clipping planes.
      We are at (vx,vy,vz) and looking towards
      the center point of the object..
      (towards (refx,refy,refz)).
   */
   perspective(450,1.00,NEARCLIPPING,FARCLIPPING);

   lookat(vx,vy,vz,refx,refy,refz,0);

}
```

200

```
/* process the popup menu selection */
processmenuhit(hititem)

int hititem;    /* item hit on the popup menus */
{
    switch(hititem)
    {
        case ROTATE:
                        break;

        case PLUSXROT:
                        rx = XROTAMOUNT;
                        break;
        case ZEROXROT:
                        rx = 0;
                        break;
        case MINUSXROT:
                        rx = - XROTAMOUNT;
                        break;
        case PLUSYROT:
                        ry = YROTAMOUNT;
                        break;
        case ZEROYROT:
                        ry = 0;
                        break;
        case MINUSYROT:
                        ry = - YROTAMOUNT;
                        break;

        case PLUSZROT:
                        rz = ZROTAMOUNT;
                        break;
        case ZEROZROT:
                        rz = 0;
                        break;
        case MINUSZROT:
                        rz = - ZROTAMOUNT;
                        break;


        case TRANSLATE:
                        break;

        case PLUSXTRANS:
                        tx = XTRANSAMOUNT;
                        break;
        case ZEROXTRANS:
                        tx=0;
                        break;
        case MINUSXTRANS:
                        tx= -XTRANSAMOUNT;
                        break;

        case PLUSYTRANS:
                        ty=YTRANSAMOUNT;
                        break;
        case ZEROYTRANS:
                        ty=0.0;
                        break;
        case MINUSYTRANS:
                        ty= -YTRANSAMOUNT;
                        break;
```

```c
        case PLUSZTRANS:
                tz=ZTRANSAMOUNT;
                break;
        case ZEROZTRANS:
                tz=0.0;
                break;
        case MINUSZTRANS:
                tz= -ZTRANSAMOUNT;
                break;


        case SCALE:
                break;


        case PLUSXSCALE:
                sx = POSSCALEAMOUNT;
                break;
        case ZEROXSCALE:
                sx = 1.0;
                break;
        case MINUSXSCALE:
                sx = NEGSCALEAMOUNT;
                break;


        case PLUSYSCALE:
                sy = POSSCALEAMOUNT;
                break;
        case ZEROYSCALE:
                sy = 1.0;
                break;
        case MINUSYSCALE:
                sy = NEGSCALEAMOUNT;
                break;


        case PLUSZSCALE:
                sz = POSSCALEAMOUNT;
                break;
        case ZEROZSCALE:
                sz = 1.0;
                break;
        case MINUSZSCALE:
                sz = NEGSCALEAMOUNT;
                break;


        case RESET:
                /* zap all values...*/
                resetallaccumulativematrices();

                break;


        case EXIT:
                exit(0);
                break;


        default:
                break;

} /* end switch */


}
```

```
/* the following routine sets all accumulative matrices to unit matrices */

resetallaccumulativematrices()
{

    unit(transacc);    /* set the trans accumulative */

    unit(rxacc);       /* set the x rotation accumulative */
    unit(ryacc);       /* set the y rotation accumulative */
    unit(rzacc);       /* set the z rotation accumulative */

    unit(scaleacc);    /* set the scale accumulative */

    /* reset all the ON values to off... */
    rx = 0;
    ry = 0;
    rz = 0;

    tx = 0.0;
    ty = 0.0;
    tz = 0.0;

    sx = 1.0;
    sy = 1.0;
    sz = 1.0;

}


/* the following routine loads a unit matrix into the input array */

unit(m)
Matrix m;
{
    static Matrix un = { 1.0, 0.0, 0.0, 0.0,
                         0.0, 1.0, 0.0, 0.0,
                         0.0, 0.0, 1.0, 0.0,
                         0.0, 0.0, 0.0, 1.0 };
    long i,j;

    /* copy the matrix elements...*/
    for(i=0; i < 4; i=i+1)
    {
        for(j=0; j < 4; j=j+1)
        {
            m[i][j]=un[i][j];
        }
    }

}


/* this routine loads a unit matrix onto the top of the stack */

loadunit()
{
    static Matrix un = { 1.0, 0.0, 0.0, 0.0,
                         0.0, 1.0, 0.0, 0.0,
                         0.0, 0.0, 1.0, 0.0,
                         0.0, 0.0, 0.0, 1.0 };

    /* load the matrix */
    loadmatrix(un);
}
```

```c
/* This section reads in the data arrays base and ground */
inputstructure(base, numpolygons, filename)
  struct threedobj base[];
  int *numpolygons;
  char filename[20];
{
    FILE *inpfl;
    int i, j;
    int polygons;
    inpfl = fopen(filename, "r");
    fscanf(inpfl, "%d", numpolygons);

    for (i=0; i<*numpolygons; i++)
    {
        fscanf(inpfl, "%d", &base[i].numpts);

        for (j=0; j<base[i].numpts; j++)
        {
            fscanf(inpfl, "%f%f%f",
                &(base[i].point[j][0]),
                &(base[i].point[j][1]),
                &(base[i].point[j][2]));

#ifdef TRACE
            printf("points[%d][0]=%.f  points[%d][1]=%.f  points[%d][2]=%.f\n",
                j,base[i].point[j][0],
                j,base[i].point[j][1],
                j,base[i].point[j][2]);
#endif
        };
    };
    fclose(inpfl);
    };
```

```
/* Computes normal for polygon and reorders polygon points to
   counterclockwise if given in clockwise order.  ax,ay,az must
   be an interior point of polygon in order to orient the normal
   vector in correct location. */
calculatenormal(xyz, numpts, ax,ay,az)
struct threedobj xyz[];
int numpts;  /* number of polygons in the xyz array */
float ax,ay,az;   /* interior point of the whole object. */


{

     float txyz[MAXPOINTS][3];   /* temp coord hold */
     long h,i,j;   /* loop temps */
     long ncoords;           /* looping for each polygon */
     int npoly_orient();   /* direction test function */
     float v1[3],v2[3];      /* vectors used to compute the polygon's normal */
     float normalmag;        /* normal's magnitude   */
     float lightmag;         /* magnitude of the light vector */
     float normal[3];     /* tempory storage for normal */
     float v1mag,v2mag;
     double vecmag();

for (h=0; h<numpts; h++)
  {
#ifdef TRACE
     printf("\nlorient xyz[][0-2]\n");
     for (i=0; i<ncoords; i++)
         printf("   xyz[%d] %f %f %f\n",i,xyz[i][0],xyz[i][1],xyz[i][2]);
     printf("   ax,ay,az %f %f %f\n",ax,ay,az);
#endif
     /* check the number of coords in the input array */
     if(xyz[h].numpts > MAXPOINTS)
     {
         printf("LIGHTORIENT: too many coords passed to me! = %d\n",ncoords);
         exit(1);
     }
    /* orient the polygon so that its CCW with respect to the interior point */
/* this section removed tempory. will replace next quarter
     if(npoly_orient(ncoords,xyz,ax,ay,az) == 1)
     {
*/          /* the polygon is clockwise, reverse it. */
/*          for(i=0; i < ncoords; i=i+1)
          {
              for(j=0; j < 3; j=j+1)
              {
                  txyz[i][j] = xyz[ncoords-i-1][j];
              }
          }
          for(i=0; i < ncoords; ++i)
          {
              for (j=0; j < 3; ++j)
              {
                  xyz[i][j] = txyz[i][j];
              }
          }
#ifdef TRACE
          printf("lorient pts reversed\n");
          for (i=0; i<ncoords; i++)
              printf("   xyz[%d] %f %f %f\n",i,xyz[i][0],xyz[i][1],xyz[i][2]);
#endif
     }
*/
```

```c
    /* the coordinates are ordered counterclockwise in array xyz */

    /* compute the normal vector for the polygon using the first 3 vertices */

    /* compute the first vector to use in the computation */
        v1[0] = xyz[h].point[2][0] - xyz[h].point[1][0];
        v1[1] = xyz[h].point[2][1] - xyz[h].point[1][1];
        v1[2] = xyz[h].point[2][2] - xyz[h].point[1][2];

        /* compute the second vector to use in computing the normal */
        v2[0] = xyz[h].point[0][0] - xyz[h].point[1][0];
        v2[1] = xyz[h].point[0][1] - xyz[h].point[1][1];
        v2[2] = xyz[h].point[0][2] - xyz[h].point[1][2];

        /* the normal is v1 x v2 */
        normal[0] = v1[1]*v2[2] - v1[2]*v2[1];
        normal[1] = v1[2]*v2[0] - v1[0]*v2[2];
        normal[2] = v1[0]*v2[1] - v1[1]*v2[0];

#ifdef TRACE
        printf("lorient normal before mag div %f %f %f\n",normal[0],
                    normal[1],normal[2]);
#endif

        normalmag = (float)(vecmag)((double)(normal[0]),(double)(normal[1]),
                                (double)(normal[2]));

        xyz[h].normal[0] = normal[0] / normalmag;
        xyz[h].normal[1] = normal[1] / normalmag;
        xyz[h].normal[2] = normal[2] / normalmag;

#ifdef TRACE
        printf("lorient normal %f %f %f\n",normal[0],normal[1],normal[2]);
#endif


    } /* end of for h ... */
}

/* this procedure computes the vector mag for use of making the unit vector*/
double vecmag(x,y,z)
float x,y,z;

{
    double t1,t2,t3,t4,t5;

    t1 = ((double)(x)) * ((double)(x));
    t2 = ((double)(y)) * ((double)(y));
    t3 = ((double)(z)) * ((double)(z));
    t4 = t1 + t2 + t3;
    t5 = sqrt(t4);

#ifdef TRACE
    printf("vecmag t1,t2,t3,t4 %f %f %f %f\n",t1,t2,t3,t4);
    printf("vecmag x,y,z,mag %f %f %f %f\n",x,y,z,t5);
#endif

    return(t5);
}
```

```c
/* this draws the object that is passed in */
drawobject(object,numpolygons)
   struct threedobj object[];
   int numpolygons;
{
  int h,i,j ;     /* loop temps  */
  for (h=0; h<numpolygons; h++)
    {
      normal(object[h].normal);
      pmv(object[h].point[0][0],object[h].point[0][1],object[h].point[0][2]);
      for (i=1; i<object[h].numpts; i++)
    {
      pdr(object[h].point[i][0],
          object[h].point[i][1],
          object[h].point[i][2]);
    }
      pclos();
     }
}
/* This section reads in the path of the missile to be displayed */
inputlinearray()
{
    FILE *inpf;
    int i, j;
    inpf = fopen("path1.dat", "r");
    fscanf(inpf, "%d",&numberofpaths);
        for (i=0;i<numberofpaths;i++)
      { fscanf(inpf, "%d",&numberofwaypoints[i]);
      }
    for (j=0;j<numberofpaths;j++)
      {
        for (i=0;i<numberofwaypoints[j];i++)
          {
           fscanf(inpf, "%f%f%f%f",
                  &waypoints[j][i][0],&waypoints[j][i][1],
                  &waypoints[j][i][2],&waypoints[j][i][3]);
          }
      }
    fclose(inpf);
      }


/* draw the path of the missile with color shading for % observation */
drawpath()
{
  int i=0, j, redtint;

  for (j=0;j<numberofpaths;j++)
    {
      redtint = (int)(255-255*waypoints[j][0][3]);
      RGBcolor(255,redtint,45);
      move(waypoints[j][0][0],waypoints[j][0][1],waypoints[j][0][2]);
      linewidth(3);
      for(i=1;i<numberofwaypoints[j];i++)
    {
      redtint = (int)(255-255*waypoints[j][i][3]);
      RGBcolor(255,redtint,45);
      draw(waypoints[j][i][0],waypoints[j][i][1],waypoints[j][i][2]);
    }
     }
  linewidth(3);
}
```

```
/* this is file rotate.h

    It is the include file for program rotate.c
    This file holds the defines and the global variables
    for programs:
    rotate.c
    rotate2.c

*/

/* defines for the menu definition routine */

#define ROTATE 1

#define PLUSXROT 2
#define ZEROXROT 3
#define MINUSXROT 4

#define PLUSYROT 5
#define ZEROYROT 6
#define MINUSYROT 7

#define PLUSZROT 8
#define ZEROZROT 9
#define MINUSZROT 10


#define TRANSLATE 11

#define PLUSXTRANS 12
#define ZEROXTRANS 13
#define MINUSXTRANS 14

#define PLUSYTRANS 15
#define ZEROYTRANS 16
#define MINUSYTRANS 17

#define PLUSZTRANS 18
#define ZEROZTRANS 19
#define MINUSZTRANS 20


#define SCALE 21

#define PLUSXSCALE 22
#define ZEROXSCALE 23
#define MINUSXSCALE 24

#define PLUSYSCALE 25
#define ZEROYSCALE 26
#define MINUSYSCALE 27

#define PLUSZSCALE 28
#define ZEROZSCALE 29
#define MINUSZSCALE 30


#define RESET 31


#define EXIT 32
```

208

```c
/* the following defines are the amounts concatenated
   each frame if the matrix concatenation is selected
   as ON
*/

#define XROTAMOUNT 25  /* 2.5 degrees of rotation each picture */
#define YROTAMOUNT 25  /* 2.5 degrees of rotation each picture */
#define ZROTAMOUNT 25  /* 2.5 degrees of rotation each picture */

#define XTRANSAMOUNT 5.0; /* 5 units of translation in the x direction */
#define YTRANSAMOUNT 5.0; /* 5 units of translation in the y direction */
#define ZTRANSAMOUNT 5.0; /* 5 units of translation in the z direction */

#define NEGSCALEAMOUNT 0.99; /* 0.99 scale each frame if ON */
#define POSSCALEAMOUNT 1.01; /* 1.01 scale each frame if ON */


/* the following variables are set when the particular matrix
   concatenation is turned ON. Otherwise they are zero...
*/

static float tx; /* translation on in the x direction */
static float ty; /* translation on in the y direction */
static float tz; /* translation on in the z direction */

static short rx; /* rotation on in the x direction */
static short ry; /* rotation on in the y direction */
static short rz; /* rotation on in the z direction */

static float sx; /* scale on in the x direction */
static float sy; /* scale on in the y direction */
static float sz; /* scale on in the z direction */


/* some globally defined matrices for the viewing matrix computation */

static Matrix transacc; /* accumulative translation matrix */

static Matrix rxacc;    /* accumulative x rotation matrix */
static Matrix ryacc;    /* accumulative y rotation matrix */
static Matrix rzacc;    /* accumulative z rotation matrix */

static Matrix scaleacc; /* accumulative scale matrix */
```

```c
/* this is file lightdefs.c

    These routines define the materials/lights/lighting models needed...

*/

#include "gl.h"

#include "lightdefs.h"

/* set up all the materials */

initializematerials()
{
    /* make the definition calls for the materials */

    /* make the defs for the terrain */
    lmdef(DEFMATERIAL,DIRT,19,dirt);
    lmdef(DEFMATERIAL,GRASS1,19,grass1);
    lmdef(DEFMATERIAL,GRASS2,19,grass2);
    lmdef(DEFMATERIAL,GRASS3,19,grass3);

    /* make the material for where the light is */
    lmdef(DEFMATERIAL,LIGHTMATERIAL,19,lightmaterial);

}

/* this routine sets up the light for the scene */

initializelights()
{
    /* define the light */
    lmdef(DEFLIGHT,MYLIGHT,14,light);
    /* turn this light on */
    lmbind(LIGHT0,MYLIGHT);
}
/* define the lighting model */

initializelmodel()
{
    /* define the lighting model */
    lmdef(DEFLMODEL,MYMODEL,10,lmodel);
    /* turn on the model */
    lmbind(LMODEL,MYMODEL);
}
/* the following routine calls routine normal() with 3 args */

xyznormal(x,y,z)
float x,y,z;    /* input normal vector */

{

    float tmp[3];    /* array to hold the normal */

    tmp[0] = x;
    tmp[1] = y;
    tmp[2] = z;

    normal(tmp);

}
```

210

```c
/* this is file lightdefs.h
   It is the file containing the material/light/lighting model defs
*/


#define MYSHININESS 10.0   /* my value for E(mss)    */
#define LIGHTMATERIAL 9

static float lightmaterial[] = {
                       EMISSION, 1.0, 1.0, 1.0,
                       AMBIENT,  0.0, 0.0, 0.0,
                       DIFFUSE,  0.0, 0.0, 0.0,
                       SPECULAR, 0.0, 0.0, 0.0,
                       SHININESS, 0.0,
                       LMNULL
                  };

/* set up the light defs for the program */


#define MYLIGHT 10

#define LIGHTX 200.0    /* loc of the light */
#define LIGHTY 100.0
#define LIGHTZ -350.0

static float light[] = {
                   AMBIENT, 0.2, 0.20, 0.20,
                   LCOLOR,  1.0, 1.0, 1.0,
                   POSITION, 0.0, 0.707106, 0.707106, 0.0,
                   LMNULL
                };

/* define the lighting model */

#define MYMODEL 11

static float lmodel[] = {
                   AMBIENT, 0.20, 0.20, 0.20,
                   LOCALVIEWER, 0.0,
                   ATTENUATION, 1.0, 0.0000,
                   LMNULL
                };
```

```
/* setup terrain definitions */

#define DIRT 12

static float dirt[] = {
                        EMISSION,  0.0, 0.0, 0.0,
                        AMBIENT,   0.47, 0.31, 0.0,
                        DIFFUSE,   0.47, 0.31, 0.0,
                        SPECULAR, 0.0, 0.0, 0.0,
                        SHININESS, 0.0,
                        LMNULL
                    };
#define GRASS1 13

static float grass1[] = {
                        EMISSION,  0.0, 0.0, 0.0,
                        AMBIENT, 0.325, 0.775, 0.0,
                        DIFFUSE,  0.345, 0.775, 0.0,
                        SPECULAR, 0., 0.0, 0.0,
                        SHININESS, 0.0,
                        LMNULL
                    };
#define GRASS2 14

static float grass2[] = {
                        EMISSION,  0.0, 0.0, 0.0,
                        AMBIENT,  0.2549, 0.61, 0.0,
                        DIFFUSE,  0.2549, 0.61, 0.0,
                        SPECULAR, 0.0 ,0.0 , 0.0,
                        SHININESS, 0.0,
                        LMNULL
                    };
#define GRASS3 15

static float grass3[] = {
                        EMISSION,  0.0, 0.0, 0.0,
                        AMBIENT,   0.0, 0.1, 0.1,
                        DIFFUSE,  0.2549, 0.41, 0.0,
                        SPECULAR, 0.2549, 0.41, 0.0,
                        SHININESS, 10.0,
                        LMNULL
                    };
```

```
/* This is the Makefile for ppgh.c */
CFLAGS =

ALL = ppgh

all: $(ALL)

clean:
    rm -f *.o

delete:
    rm -f *.o $(ALL)

ppgh: ppgh.o rotate.h lightdefs.h lightdefs.o
    cc -o ppgh ppgh.o lightdefs.o  -Zg $(CFLAGS)

ppgh.o: lightdefs.h

lightdefs.o: lightdefs.h
```

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center   2
   Cameron Station
   Alexandria, Virginia 22304-6145

2. Library, Code 0142   2
   Naval Postgraduate School
   Monterey, California 93943-5002

3. Cruise Missiles Project (PMA 281)   2
   Naval Air Systems Command Headquarters
   Washington, DC 20361-1014

4. Neil C. Rowe, Code 52   2
   Department of Computer Science
   Naval Postgraduate School
   Monterey, California 93943-5100

5. LT David H. Lewis   1
   Commanding Officer
   Naval Surface Force Pacific
   Readiness Support Group
   San Diego, Box 124
   Naval Station
   San Diego, California 92136-5126

6. Maj. Lawrence R. Wrenn   2
   Commanding Officer
   Marine Corps Tactical System Support Activity
   Camp Pendleton, California 92055

7. Commandant of the Marine Corps   1
   Code TE 06
   Headquarters, U.S. Marine Corps
   Washington, D.C. 20360-0001