2-8-1989

Real time maze traversal

Robert Spina

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation
Rochester Institute of Technology
College of Engineering

Real Time Maze Traversal

by

Robert Spina

A Thesis, submitted to
The Faculty of the College of Engineering,
in partial fulfillment of the requirements for the degree of
Master of Science in Electrical Engineering

Approved by:

Roy Czernikowski, Professor and Committee Chairman

J. A. Biles, Associate Professor

8 February 1989
I would like to thank the Department of Computer Engineering for the use of its facilities and under whose direction this work was completed. I would also like to thank the Department of Electrical Engineering for allowing me the flexibility they have shown throughout the completion of my degree. Above all, I would like to thank my committee chairman, Professor Roy Czernikowski, whose enthusiasm for this project helped me complete the tedious parts.
ABSTRACT

This thesis investigates the application of artificial intelligence to real time control in combination with image analysis techniques. The system consists of a "mouse" and a "cheese" in a two dimensional maze which are viewed by a video camera connected to a frame grabber board in a personal computer (PC). The maze is designed to be alterable during the mouse's navigation towards the "cheese" or target. The PC runs the frame grabber software which analyzes the maze to find the cheese, mouse and movement options during traversal. Parameters about the mouse's position, the maze constraints in the vicinity of the mouse and the general direction of the target are passed to a Prolog search algorithm which makes decisions on the next course of action for the mouse. Parameters on the movement of the mouse are passed via a serial port to a single board computer which controls the mouse's motion with two stepper motors. The intent is to investigate the integration of a simple real time control system with an intelligent host before expanding the system to handle more complex control algorithms where the computational requirements could easily overpower a single processor. This approach allows the possibility of mixing and matching processors for different applications. It also becomes easier to determine whether the AI-image analysis or the real time control activities are the computational bottlenecks and to decide on appropriate corrective measures.
Table of Contents

1.0 Introduction and Background .................................. 1
  1.1 Problem Statement ........................................... 3
  1.2 System Constraints .......................................... 3
2.0 Functional Specification ........................................ 6
  2.1 User’s Manual ............................................... 6
      2.1.1 Maze Construction .................................... 7
      2.1.2 Hardware Configuration ............................... 8
      2.1.3 Runtime Initialization ............................... 11
      2.1.4 Maze Traversal ...................................... 15
3.0 Software Architecture ......................................... 17
  3.1 Module Design ............................................... 21
      3.1.1 Prolog Search Algorithm ............................ 21
      3.1.2 Image Processing ..................................... 22
      3.1.3 Motion Control ....................................... 23
      3.1.4 Knowledge Display .................................... 24
  3.2 Data Base .................................................... 25
  3.3 Communication Among Modules ................................. 26
4.0 Verification and Validation .................................... 29
  4.1 Test Plan ..................................................... 29
  4.2 Deliverables ................................................ 30
5.0 Conclusions .................................................... 32
  5.1 Problems Encountered and Solved ............................. 33
  5.2 Remaining Limitations ....................................... 34
  5.3 Suggestions for Future Extensions ........................... 36
6.0 Appendicies ...................................................... 39
    Appendix A TRAVERSE.BAT .................................... 39
    Appendix B Setting up the National DB32016 .................. 41
    Appendix C Software Listing for COMMON.PAS ................. 43
    Appendix D Software Listing for STARTUP.PAS ............... 54
    Appendix E Software Listing for RUNTIME.PAS ............... 58
    Appendix F Software Listing for SERIAL.PAS ................. 63
    Appendix G Software Listing for KNOW.PAS .................. 70
    Appendix H Software Listing for AUTO.PRO .................. 75
7.0 References ...................................................... 85
1.0 Introduction and Background

Rising production costs and the introduction of complicated machinery into the workplace have made industry interested in intelligent machines. These are machines that can cope with the variables of real life. Reaction to changes must be accurate and timely.

The advances in computer hardware and software over the past 20 years has sparked a new interest in real time artificial intelligence. The merging of "slow" thinking programs and "fast" controllers is a growing area of interest. Where do the bottlenecks in the system occur? Can systems deal with unexpected changes in their operating environment?

The main area of interest is what happens between the time a computer gets a problem and when it effects a solution. This is the area where plans of action are generated and carried out. There must be a time map in which past events are stored when we work with computer plans.

This time map allows the computer to backtrack or in some instances learn from previous decisions. This time map represents the knowledge of the system. It is critical for a planning system system to have this type of information available to it.

"The design of a complete planner would have to address the following issues:

1) What is the correct notation for plans?
2) How are time maps produced and maintained?
3) How can problems be anticipated and corrected?
4) How does one manage to search through the space of possible plans?
5) How are plans translated into action in the real world?
6) How does one monitor the progress of a plan?
7) How does one re-plan when things go wrong?

The most difficult of the problems is number 6. This process is called execution monitoring and may require a great deal of computer resources. For real time applications, this may be helped with better sensors and effectors.
Adaptive navigation algorithms for mobile robots is one area where computer plans can be used. Jorgensen [1] mentions that robot navigation can occur in three environments: a static known environment, a static unknown environment, and a dynamically changing environment. The third is the most difficult.

Tominaga and Bavarian [2] contend that the problem of robot navigation through an unknown environment can be reduced to the problem dealing with a known environment with one basic assumption. This assumption is: All unexplored regions are free from obstacles. With this assumption, one can use any of the existing techniques to compute the optimal path.

The representation of terrain knowledge is always a problem. Gex and Campbell [3] take the approach that it is just as important to keep track of the free space as it is the obstacles. This helps to minimize the amount of information passed to the planner. They also partition the motion tasks to a separate Locomotion module.

Navigation algorithms make an underlying assumption that the destination or goal of the robot is known. Determining this goal is not the task of the planner. The planner is used to create subgoals that will eventually lead to the robot reaching its final goal.
1.1 Problem Statement

This thesis is intended to provide a basis for the exploration of the problems associated with developing intelligent machines. Its primary applications are in the field of robotics. Robots being real time machines that require gathering sensory information, interpreting this information and effecting physical actions in changing environments.

In order to demonstrate these concepts, we will use the classic problem of a mouse in a maze. One interesting aspect of this project is that the maze may be altered during traversal. Another is that there is no foreknowledge of the maze given to the mouse. What exists initially is a search strategy and the task of finding a piece of cheese located in the maze.

By incorporating some image analysis into this prototype along with the possibility of a randomly changing environment (changing maze geometry during runtime), the system allows the exploration of more robust robotic applications. Such future robotics investigations may require significant changes to the software on the low level real time controller and to the image recognition software. The approach would allow an orderly investigation of more complex systems from the experiences gained in operating this prototype. The system architecture also allows for the convenient replacement of the low-level control processor with some other or a network of others.

1.2 System Constraints

The main purpose of this work is to explore the development and integration of intelligent programs with real world hardware. The development of complex robotic devices is outside the scope of this project. The concentration is placed on developing intelligent machines, not sophisticated motion control. For these reasons an abstract rendering of a mouse and maze was chosen to demonstrate this work.

Several alternatives were considered in selecting an appropriate mouse and maze. Originally a formal walled three dimensional maze was proposed. However, real estate restrictions and the complexity of the mechanical mouse needed to traverse such a maze made this approach prohibitive.

Ideas on how to get movement choices out to the mechanical mouse ranged from an umbilical serial link to an infra-red remote link. The complexity of these approaches and the
problems they introduced into other parts of the system forced them to be eliminated.

In an attempt to simplify the maze and mouse construction, the maze was transformed from three to two dimensions. This maze could be represented by reflective strips laid out on some flat surface. This approach provided a clean path for communication with the mouse. It also made modification of the maze during traversal easier. New tape walls could be added and old ones removed in a minimum amount of time.

However, with this setup, the real estate problem with the maze still existed. In addition, the mouse would still have to be very complicated to insure adequate motion quality.

The maze finally selected is very much like the maze given in children's puzzle books. You start somewhere and find your way out. Even this has been simplified to restrict the maze to orthogonal corridors and walls. See Figure 1.

![Two Dimensional Maze Diagram](image)

Figure 1 - Diagram of Two Dimensional Maze

Given these limitations the mouse can be modeled as the intersection of two crosshairs. In actuality, the mouse is located beneath the maze. This eliminates any interference problems with the mouse and maze walls.

A good conceptualization can be made by imagining an ETCHA-SKETCH drawing toy. In this toy, a small center piece located under the surface of the viewing glass wipes away
metal shavings to draw the outline of a picture. The direction of the center piece is controlled by two knobs located on the left and right side of the toy. The user manually turns these knobs to get a sketch. The knobs are replaced with stepper motors and the direction requests will be issued by a single board computer for this project.

Sensory gathering was the next problem to be solved. How will the mouse tell if there is a wall or an opening in the maze? The two senses available to humans for this purpose are sight and touch. With the above physical implementation, the only viable means of wall detection was visual. Touch was discarded because no physical walls existed.

A small reflective sensor might have been used to detect a wall or corridor. However, this sensor would have had to rotate 360 degrees in order to provide the mouse with movement information in front, in back, to the left, or to the right of it.

This scheme also eliminated the interesting possibility of letting the mouse have one more sense in addition to sight. This is the sense of smell. In order to simulate the sense of smell, the mouse or some other part of the system must know where the cheese is located. The use of smell in maze traversal is explained in detail in Section 3.1.1 - Prolog Search Algorithm.

A video camera is used to provide both the mouse's sight and smell. Image processing algorithms determine the existence of walls and corridors immediately surrounding the mouse. The system can then superimpose an imaginary image of the cheese on a video monitor showing the aerial view of the maze, with the location of the cheese having been selected by the user. This information is hidden from the mouse. Only the relative distance from the mouse to cheese is known by the mouse, and this relative distance is used as the mouse's sense of smell.
2.0 Functional Specification

The maze consists of a logical 32 x 64 grid pattern (see Figure 2). This is similar to a chess board. Given a preconstructed maze, cheese location and starting mouse location the system will have the mouse find the cheese. The mouse will be presented with possible moves and will select one based on the "best smell" strategy. This search strategy is explained in detail in Section 3.1.1 - Prolog Search Algorithm.

![Diagram of Logical Maze](image)

**Figure 2 - Diagram of Logical Maze**

This choice is passed to the hardware that will physically move the mouse through the maze. This move will be verified and new possible moves determined. Additionally, the mouse will have to verify and maintain the integrity of its knowledge base. This is required since the maze is allowed to change during transversal. This process is continued until the cheese is found.

2.1 Users' Manual

The following sections describe how an individual would setup and use the system to perform a maze transversal. If one is starting from scratch, that is no maze exists, the user would have to perform the tasks outlined in Section 2.1.1 of this document. Generally it is assumed that a maze already exists and the user wants to modify the existing maze or simply
traverse it with different starting mouse and cheese locations. This assumption is made due to the lengthy process of building a maze from scratch. However, the process and constraints of building a maze indicated in Section 2.1.1 - Maze Construction must still be maintained.

2.1.1 Maze Construction

The physical maze is 24 x 24 square inches. See Figure 3. It is constructed out of a piece of Plexiglas and an equally sized piece of plywood. The plywood serves as the base. The Plexiglas is separated from the plywood with eight 6 inch standoffs. It is on this piece of Plexiglas that the maze walls will be laid out.

The maze is constructed using manila colored masking tape on the top surface of the Plexiglas. It is very important that the masking tape be of the proper color. This is because the maze walls must lie within the 64 to 192 eight-bit gray scale region of the frame grabber used to provide the sense of sight.

The maze walls must be made a multiple of 0.5 inches in width with the maze corridors being 0.5 inches. The maze walls must be orthogonal. This is required so that the logical maze will overlay the physical maze properly. This constraint will be made more evident in the runtime set up of
the maze. This process is simplified with the use of a 24 x 24 square inch quadruled template with 0.5 x 0.5 square inch boxes. This template may be placed under the Plexiglas as a guide for the user when building the maze. It must be removed before runtime set up.

Maze walls should continue to be laid out until the entire maze is completed. See Figure 4 for an example maze.

![Maze Diagram]

Figure 4 - Completely Laid Out Maze

2.1.2 Hardware Configuration

The following is a description of the hardware components, their configuration and some insight into their role in the system.

(1) Personal Computer Configuration:

The personal computer used is an IBM PC-AT with 30 Mbyte hard drive and 640 Kbytes of RAM. A 64 KByte section of the available RAM has been partitioned for a RAM Drive. This is to speed the passing of parameters between the PASCAL image processing software and the Prolog search algorithm.

(2) Frame-Grabber - including TV camera and mounting:

The frame grabber is a DataCube model 26-00138. This card has been placed into one of the available
IBM PC-AT bus slots at location 80000 hexadecimal. This device provides a logical array of 512 x 512 picture elements (pixels). They must be addressed as four banks of 128 x 512 pixels. Attached to this board is an RCA charge coupled device (CCD) solid state video camera which is mounted on an eight inch tripod. This camera will be used to capture an overhead view of the maze and mouse. It should be placed approximately three feet from the center of the physical maze.

(3) Single Board Computer:

The single board computer is the National Semiconductor DB32016. The DB32016 communicates with the PC via a RS-232-C serial link. It is the responsibility of this card to translate motion choices into physical movement.

(4) Mouse and Maze Description:

As mentioned in Section 2.1.1 - Maze Construction, the maze is constructed out of a piece of 24 x 24 square inch Plexiglas and an equally sized piece of plywood. The plywood is covered with a 1/4 inch thick piece of foamcore painted flat black.

The Plexiglas is separated from the plywood by six inch standoffs. Each of these standoffs will contain a pulley to help guide the supporting string used to translate the stepper motor motion into the proper north, south, east or west direction. Attached to the strings will be two rods. These will be orthogonal to each other. One will run north-south the other east-west. Where they intersect will be 0.25 by 0.25 square inch light emitting diode (LED). This LED represents the mouse. The mouse is moved just below the Plexiglas by the set of pulleys and supporting strings.
2.1.3 Runtime Initialization

Once the maze has been constructed and the hardware properly connected, the user is ready to execute the runtime software. This is done by executing the MS-DOS batch file named TRAVERSE.BAT located in the root directory of the IBM PC-AT. This file will load the necessary modules into the virtual disk and start the execution of the Prolog search algorithm. A copy of the file can be found in appendix A.

There are three remaining tasks to be performed before free running traversal may begin.

1) Alignment of physical maze and the TV camera.
2) Cheese placement.
3) Initial mouse location.

The first involves the alignment of the physical maze and the camera so that they correspond to the logical grid elements. Initially the user will see the logical grid pattern overlaid on the image of the physical maze located on the video monitor. The user must make sure the logical grid locations are matched to the physical maze located on the video monitor. The user will be queried if the current setup is satisfactory. If it is not, the user must adjust the camera position until it is satisfactory.

Figure 6 - Maze alignment Screen
The second is the placement of the cheese icon. The cheese will only exist in software. It will appear on the video.
monitor attached to the frame grabber. The user is asked where the cheese is to be placed. The system is looking for the XY coordinate pair that corresponds to one of the 32 x 64 logical grid elements of the maze. The location (1,1) is in the upper left corner of the maze. The X axis extends down in the vertical direction. The Y axis extends right in the horizontal direction. The user will be queried if the selected cheese location is satisfactory. [NOTE: The cheese location is not allowed to change during transversal.]

Figure 9 - Cheese Icon Placement Screen

Figure 10 - Video Monitor with Cheese Icon
The third and final input from the user is the starting location of the mouse. Movement of the mouse is accomplished with the "N", "S", "E" and "W" keys on the IBM PC-AT keyboard. The capital keys (N,S,E,W) will make the mouse move in large steps. The lower case keys (n,s,e,w) will move the mouse in smaller increments. Continue moving the mouse until it is in a satisfactory location. This part of the setup is terminated by typing in a lower case "q".

Figure 11 - Mouse Placement Screen

Figure 12 - Video Monitor After Initialization
The user will not have to provide any other inputs unless the maze is to be altered during traversal.

2.1.4 Maze Traversal

The mouse is ready to traverse the maze in search of the cheese at this point. Depending on the interest of the user, he/she may view:

1) PC Screen
2) Video Monitor
3) Physical Maze

On the PC screen, the user will be able to graphically see the maze knowledge grow. The user will be able to see the conflicts in the knowledge that arise during the traversal of a changing maze. The screen will be divided into 32 x 64 locations representing the logical grid of the maze. Initially the screen will be black showing no knowledge of the maze except for an icon at the starting cheese and mouse locations. A green square will be placed in the corresponding screen locations wherever a possible move exists. This means that if the maze never changes the user will only see black walls and green corridors of tried or possible paths and the physical mouse’s location.

If, however, in backtracking the knowledge base is found to be corrupt, the user will be able to see this as well. The maze locations that were walls the first time through and now have become openings will be changed from black to yellow. The locations that used to be openings will be changed from green to red. This allows the user to get a feel for what the mouse is thinking. See Figure 13.

The video monitor will be displaying the maze as seen by the image processing software. This screen allows the user to see the correlation between the physical world of human perception and that of limited sensory perception.
Figure 13 - Display of Knowledge Base on PC Screen.

The physical maze is of interest to see the translation of desired location to actual movement. The user will also be able to notice the adjustments that are required after each move. These adjustments are done to guarantee that the move was performed properly so that the next iteration of movement selection will not be at a disadvantage.

It is important to note that these adjustments are typical of the problems found in dealing with the real world. They are critical in the ability of the planner to minimize the possibly catastrophic affects of items 3, 5, and 6 outlined in Section 1.1 - Problem Statement.

During traversal the user may choose to alter the maze. It is desirable to perform changes during the display of the knowledge base on the PC screen. This is due to the rather lengthy process of displaying this information. It also serves to limit the type of changes the user would want to perform to simple opening or closing of corridors. This helps the user maintain the physical maze constraints outlined in Section 2.1.1 Maze Construction.
3.0 Software Architecture

In order to clarify the process involved in a maze traversal, it is helpful to look at the macroscopic flow of the system's programs. Shown in Figure 14 is a high level flow chart of the system.

Figure 14 - High Level Flow Chart
In this system, it is very natural to divide the tasks into two functional areas. The first being real time control and the second being the intelligent supervisor. Similarly it is convenient to divide these tasks on to two separate computers. Ideally, one of the computers would only be responsible for making decisions on how to achieve the goal of finding the cheese. The other computer would take care of the details of translating decisions into the real world. The PC in this system will be the intelligent supervisor and the National Semiconductor DB32016 will be the real time controller.

There are four major modules in the system.

1) Prolog Search Algorithm
2) Image Processing
3) Motion Control
4) Knowledge Display

Each of these modules are discussed in detail in Section 3.1 - Module Design. For the time being it is sufficient to know that all of these modules must be orchestrated very closely since there are many cross dependencies in the modules.

The following four figures are the structure diagrams for the system. The details regarding each function or procedure can be found in the source listings in appendicies C,D,E,F,G.

Figure 15 - Structure Diagram for Auto.pro

18
Figure 17 - Structure Diagram for Runtime Image Processing
3.1 Module Design

The major software modules are briefly mentioned here and will be described in detail in the following sections:

1) Prolog Search Algorithm:

This module is implemented in Borland's Turbo Prolog. It can be thought of as the brains of the system. It evaluates options for the mouse movement, verifies knowledge of the maze and issues movement selection.

2) Image Processing Algorithms:

The image processing software is implemented in Borland's Turbo PASCAL. There are two major modules that fall into this category: Startup and Runtime. Both have the responsibility of locating the mouse and walls in the maze. The Runtime portion must also act as the closed loop feedback on mouse positioning.

3) Motion Control:

Motion control is implemented in the "C" programming language. This module issues the necessary control signals to the stepper motors that will physically move the mouse.

4) Knowledge display:

This module is implemented in Borland's Turbo PASCAL. The knowledge of the mouse is displayed for the user. This graphical representation shows movement options and knowledge corruption.

3.1.1 Prolog Search Algorithm

It is the job of the search algorithm to make decisions for the mouse movement. Initially, the system has no knowledge of the maze. Information about the maze grows as it is being traversed. The maze appears as a 32 x 64 logical grid to the search algorithm. Movement options will be presented in integer distances of these logical grid steps.

The Prolog search algorithm must add to its knowledge of the maze as the mouse "sees" new walls and options. It must also prune paths that have been taken or changed. When options for a location are passed to the search algorithm, they are checked against any currently existing knowledge of that
location. The knowledge base is modified if any discrepancies exist.

Given the options at the maze location, a move will be selected. That choice will be pruned from the options database. If the mouse reaches a dead end without finding the cheese, or if a path gets "significantly" worse with respect to a proximity "smell" metric (which will be defined later), the mouse will, at least temporarily, abandon the current path in favor of backtracking to the previously found "best" location (i.e. "strongest smell" at the closest backtrack distance) and resume its quest from that point. If in backtracking, however, the mouse finds the maze modified from its previous navigation through a region, all previously stored information about paths emanating from the changed traversal and option selection will proceed according to the "rules".

This "smell" metric is simply the square of the Euclidian distance of a maze location to the cheese location, the square of the distance is used to simplify the computations in Prolog. Therefore, the smell metric \( S(X_p,Y_p) \) is calculated for a given maze position \((X_p,Y_p)\) to the cheese position \((X_c,Y_c)\) as \( S(X_p,Y_p) = (X_p-X_c)^2 + (Y_p-Y_c)^2 \).

Note that since \( S(X_p,Y_p) \) is implemented as a routine, it is easy to eliminate the search for "rule" seeking the strongest smell by flattening this metric, i.e. equalizing \( S(X_p,Y_p) \) for all \( X_p \)'s and \( Y_p \)'s.

The attempt is to not burden the search process with the complications of real motion. All maze traversal decisions are made at the abstract level of the logical grid. Therefore, the search algorithm will simply issue its movement choice as either a step North, South, East, or West. Physical world problems will be dealt with by the real time controller and image processing modules.

3.1.2 Image Processing

The start up image processing module takes a snapshot of the entire maze using the frame grabber. The colors used for the maze corridors and walls as well as the mouse have been chosen so as to place their images into unique areas of the frame grabber grey scale. This allows the image processing software to use simple thresholding to locate possible sites of the mouse in the maze. Since no knowledge of the maze exists initially, a brute force search of the maze is performed. The software starts in the upper left corner and proceeds as if reading a book.
Once a possible site has been located (i.e. proper grey scale value for mouse), a more thorough analysis of the site is performed. A statistical template of the mouse is used. If the template match fails, then the software will look for other possible sites. If the match is successful, this position is translated into one of the logical grid locations of the maze.

The runtime image processing performs recognition of maze walls and verification of mouse position. This occurs during traversal. The system at this point knows the location of the mouse. It then takes a look at the positions to the North, South, East, and West. This is done by counting the number of wall colored pixels at these locations. If enough of the pixels meet the criteria, then the location is declared wall.

3.1.3 Motion Control

The single board computer, will issue commands to the stepper motors. This code is executed on the National Semiconductor DB32016 computer. Once a command such as "N" is received, the software will issue the appropriate sequence of pulses to the steppers. It will then issue a "D" back to the PC to indicate that it is done. At this time, the image processing software will use the template matching scheme to verify the mouse location. Special tweaking commands will be issued to the mouse controller if the mouse is not found in the correct location. This cycle is completed until the mouse is properly placed.

There is a centering algorithm performed after each move of the mouse. Ideally the mouse would have its center coincide with the center of the current logical grid location. However, due to slippage and possible missed steps the mouse does not always end up centered. To compensate for this, the logical grid location is divided into four equal quadrants. There is an upper left, upper right, lower left, and lower right quadrant of the original grid location.

Each of these quadrants has twenty five percent of the pixel count allotted to a logical grid location. The skew of the mouse can be determined by counting the number of mouse colored pixels in each of the quadrants. A perfectly centered mouse would have twenty five percent of the total 128 pixels mouse colored. The centering algorithm must be called when this is not the case. See Figure 18.

The first centering is performed North-South. Small or "micro-steps" are issued to the stepper motors by the mouse controller until the mouse pixel count in the two upper
quadrants are within ten percent of the lower two quadrants. The same algorithm is performed East-West. In this case, the micro-steps are issued until the mouse pixel count in the two right quadrants. This centering algorithm is critical to reduce the cumulative motion error in the system.

![Logical Grid, Mouse, Wall]

WEST - EAST

NORTH - SOUTH

Figure 18 - Skew Detection of Mouse

3.1.4 Knowledge Display

This module is one of the most useful to the user. It allows a view into the mouse’s world of limited perception. This module maintains its own knowledge of the mouse and maze. Before the movement options are passed to the Prolog Search Algorithm, the knowledge display module updates and displays its information.

The information is kept in a two dimensional array. This array represents the logical grid of the maze. The movement options are read in and the array updated.

Updating involves an algorithm to properly select the grid colors defined in Section 2.1.4 - Maze Traversal. This implies there is some redundant knowledge base verification in this module and the Prolog Search Module. However, the corrupt knowledge is only displayed here. There is no attempt to correct the knowledge base within this module.
The updated maze knowledge is then displayed on the screen of the personal computer. It provide the user with a sense of progress in the system. After a predetermined time interval, the system moves on to the Prolog Search Module and the screen is cleared. This is a limitation of the system calls performed in Turbo Prolog.

3.2 Data Base

The data base elements used to facilitate the search algorithm are:

1) current_mouse(X,Y)
2) cheese(X,Y)
3) options_at(X,Y, direction, distance, pruned)
4) how_i_got_to(X,Y, direction, distance)

Their use is outlined below:

**current_mouse(X,Y):**

This element provides the current location in the logical maze grid of the mouse. It is necessary to calculate the "smell" metric discussed in Section 3.1.1 Prolog Search Algorithm.

**cheese(X,Y):**

This element contains the information given by the user at runtime initialization as to the placement of the cheese in the logical maze grid. It is necessary to calculate the "smell" metric discussed in Section 3.1.1 Prolog Search Algorithm.

**option_at (X,Y, direction, distance, pruned):**

This allows the mouse to keep track of the current or previous options for a given maze location. X and Y are the coordinates of a known location. Direction is n, s, e, or w depending on which way the mouse is looking. Distance is the integer number of steps the mouse may take in the indicated direction. Pruned is the flag used to let the mouse know if it has already been down a path. A "t" indicates tried and a "u" indicates未成.
how_i_got_to(X,Y, direction, distance):

This element is used to retrace steps through the maze. X and Y are the coordinates of a previously visited location in the logical grid. Direction indicates which path (North, South, East, or West) was taken to arrive at X,Y. Distance is the number of integer steps taken in this direction. This structure is required because it is not enough to know just the previous locations. The mouse must know the actual direction and distance used so that it can retrace its steps and verify its knowledge base.

3.3 Communication Among Modules

The sharing of information between modules is perhaps the most confusing element of the software architecture. Even though small amounts of very simple data are involved. Most of the modules require a mixed subset of the total data generated. The use of multiple languages complicates this issue further.

Because real time constraints are an issue, the amount of redundant computation is minimal. During traversal the system memory is destroyed when switching between some of the modules. This happens when executing a module or some portion of a module, that has source code written in a different language from the currently executing module.

An example of this occurs when switching control from the Prolog Search Module to the Image Processing Module. Since control is eventually returned to the Search Module, the Prolog environment must be saved and restored. This is done automatically by some system calls in Turbo Prolog. However, this means that data can not be passed as parameters. The only solution is to store the data in files. This scheme keeps a permanent record available to all modules.

Data types such as integer and real differ in format from Turbo Prolog to Turbo PASCAL. Therefore, data is converted to an ASCII character string before it is stored. The information will then have to be converted back to an integer when read by any module.

The ASCII files generated at runtime are:

1) cheese.dat - This file contains the user selected cheese location. The data is stored as an ordered pair representing a logical grid location.
2) mouse.dat - This file contains the initial mouse location. The data is stored as an ordered pair representing a logical grid location. The value of this location is determined by the Image Processing Module and is used by the Search Module as a starting reference.

3) was.dat - This file contains the current (before move) logical grid location of the mouse. The Image Processing Module uses this information to limit the search space for the movement options and centering of the mouse.

4) option.dat - This file contains the number of steps available to the mouse for a given location. The order of the steps are North, South, East and West. Currently the mouse is only able to "see" one step in any direction. If a wall exists, the value in the file will be 0. This information is generated by the Image Processing Module and is used by the Prolog Search Module.

There is one more file created at runtime. This is "know.pic" it is a Turbo Pascal integer file. It contains the information the Knowledge Display Module requires to draw the mouse's knowledge of the maze on the PC screen. No other module uses this information.

The single board microcomputer receives its commands via an RS-232C serial interface. This information is sent by the Prolog Search Module and represents the direction the mouse is to move. This information is also in ASCII format with the following meaning:

```
N - Large step North
S - Large step South
E - Large step East
W - Large step West
n - Small step North
s - Small step South
e - Small step East
w - Small step West
```

When the single board computer believes it has completed a move it will send back an ASCII "d" for done. The Image Processing Module waits until receiving this character and then confirms the move. If any "tweaking" is required it will issue one of the above movement commands until it is satisfied with the results.
A visual aid for this data sharing is shown in Figure 19. The bubbles represent conceptual processing states and the rectangles represent files. The direction and labeling of the connecting lines help show the flow of data in the system.

Figure 19 - Data Flow Diagram for System
4.0 Verification and Validation

This section contains an outline of the intermediate and final milestones used to complete this project. Breaking the project into reasonably sized pieces greatly simplified the system integration task as well as provided a greater understanding of each module.

4.1 Test Plan

The following are the milestones used to build the prototype of this thesis, although they will not necessarily be accomplished in the order presented.

1) Prolog Search Algorithm

Plan: Write Prolog algorithm that will execute on software simulated maze developed on the frame grabber.

Acceptance: Visible confirmation of a software mouse icon traversing simulated maze on the video monitor.

2) Variable passing between Turbo Prolog and PASCAL.

Plan: Write Prolog routine to convert a single digit integer to its ASCII character and store it in a virtual disk file.

Acceptance: Write PASCAL module that prompts the user for initial cheese location in integer form and convert it into ASCII and store it in a file on the virtual disk. This file is then read by a Prolog input routine and displayed on the PC screen. A similiar test is to be constructed to get from Prolog to PASCAL.

3) Location and Quantization of mouse in maze using image processing software.

Plan: Develop PASCAL routine that performs statistical template match of mouse to screen location. This routine will color in the logical grid space on the video monitor it believes the mouse occupies.

Acceptance: Visual confirmation on the video monitor that the appropriate logical grid location is colored.
4) Motion control of stepper motors through terminal connected to the National Semiconductor DB32016.

Plan: A "C" language input routine written to request a direction (n,s,e,w) executing on the DB32016 will issue the corresponding pulses to the stepper motors.

Acceptance: Visual displacement of mechanical mouse under the maze surface will confirm stepper motor control.

5) Communication via RS-232-C serial link between PC and National Semiconductor DB32016.

Plan: A PASCAL input routine resident on the PC requesting a direction (n,s,e,w) will issue this command via RS-232-C to the DB32016. The DB32016 will then issue a done "D" command to the PC.

Acceptance: Visual displacement of the mechanical mouse under the maze surface will confirm communication between the PC and the DB32016. The PC will display a completed message once it receives the done command from the DB32016.

6) Integration of system components for automated maze traversal.

Plan: Take building blocks of above five items and produce an automated system.

Acceptance: Committee members visual confirmation.

4.2 Deliverables

The following are the deliverable items for this thesis.

1. Hardware maze that can be interfaced with the National Semiconductor DB32016 computer card.

2. Turbo Prolog search algorithm source code.

3. Turbo PASCAL image processing software source code that locates mouse and maze walls. Implemented for use in conjunction with DataCube frame grabber.

4. GNX "C" source code for motion control of maze stepper motors.
5. PASCAL source code used to display the knowledge base of the mouse on the PC screen.


7. Demonstration and presentation to the thesis committee as well as to the RIT community.
5.0 Conclusions

This thesis was intended to explore some of the problems associated with intelligent machines. It becomes obvious that our knowledge of intelligent real time control has barely begun to scratch the surface of problems that face industry. It is clearly a topic of great interest as automation is introduced into factories.

One important point to be made is that the full impact of the real world constraints would not have been felt if the maze had been simulated in software. Physical systems are a must. They provide insight into the problem that can not be gained thru simulation.

An interesting case in point is the physical motion of the mouse. The reaction of the pulleys and guide wires could only be realized with a physical system. The translation from what is a simple motion concept to the Prolog Search Module to the problem of centering and mouse placement is enlightening. The complexities of the real world outside of the maze are staggering.

Another issue that becomes apparent is that this is a very multi-disciplinary field. All types of expertise are needed. Each of the pieces in this work: Search Module, Image Processing, Motion Control could themselves be complete research topics. Additionally, the aspect of system engineering these components to function together is nontrivial. Some of the most difficult skills are learned here.

The original perception of "slow" thinking controllers being the bottleneck of the system was not demonstrated in this work. The Prolog search algorithm was one of the fastest modules of the system. Admittedly the task was not that complicated. However, it would require substantial increases in complexity for this module to be the bottleneck.

The improvement of sensors and effectors would greatly ease the problems of intelligent real time control. This area is behind our present ability to simulate intelligent choices with a computer. As we have seen, the "brains" are hampered by these two factors. Fortunately, a great deal of work is being done in this area.

Particularly helpful in this project was the use of the graphical knowledge display in understanding the complexities of the project. By choosing to look at only this information the user is able to get a very real feel of the task at hand.

A human cannot help but cheat when viewing the physical maze.
This is because they see the entire solution. This hides the complexity of movement selection. By limiting the view of the maze, choices that would normally seem foolish now become perfectly legitimate. A human in the same circumstances with the same limited knowledge would make no better choices than the mouse.

5.1 Problems Encountered and Solved

Some of the problems have already been outlined in section 1.2 - System Constraints. In addition the following are noteworthy.

1) Selection of Maze Background, Mouse and Wall Colors

The maze background, mouse and wall colors were chosen to occupy distinct regions of the Frame-Grabber grey scale. This was done to reduce the complexity of the image processing. An obvious choice is to select regions at the two extremes and one in the middle. This gives us black, white and grey. An arbitrary assignment of these colors is not sufficient. A significant improvement in performance was found by making the mouse white and the maze background black. Since the maze has depth, there were stray grey levels produced on the background by shadows from the other hardware. Painting the background and anything below the maze surface flat black eliminated this problem.

2) Set Up Procedure

The proper building and aligning of the maze and video camera are critical. Repeatability was almost impossible before the setup and test procedures of Section 2.1 - User's Manual. Ensuring a good start will eliminate or simplify many of the problems encountered during execution. This seems to be a characteristic of many of the systems in this field.

3) Centering

The motion quality of the physical maze turned out to be poor. The slack in the guidewires and pulley configuration made it impossible to guarantee mouse location based solely on stepper motor control. All that could be counted on that the mouse was probably in the general neighborhood of the desired location.
The centering algorithm was developed to solve this problem. It provided the necessary closed loop feedback between the physical maze and the motion control module. The specifics of this algorithm are outlined in Section 3.1.3 - Motion Control.

4) Physical maze construction

This is one aspect of the project that took substantially more time than expected. Conceptually the problem of the physical maze is simple. However, actually building the hardware proved, if not difficult, frustrating.

Perhaps this could have been made easier with the help of mechanically proficient individuals. This cross-discipline problem strongly emphasises the need for many diverse experts when undertaking any other sizable project.

The maze was finally made workable with patience and many attempts. It has held up remarkably well.

5.2 Remaining Limitations

Listed below are the limitations that constrain the performance and behavior of the system. Stating these limitations will help the user understand the behavior of the system. Future extensions of the project may eliminate some or all of them.

1) One Step Look Ahead

The vision of the mouse is currently limited to one logical grid location North, South, East and West. This was done to simplify the image processing software necessary for wall detection.

This means that the mouse does not know how long a corridor is. This extremely myopic vision does eliminate some of the possible search strategies. For instance, the mouse would not know if the cheese is straight ahead, but just out of sight. However, the use of the best smell strategy helps in making the proper movement choices.
2) Backtrack to Last Acceptable Location

The two possible backtracking possibilities are:

a) Backtrack to first step of wrong path
b) Backtrack to last acceptable step

In the first method, the mouse would make movement selections until it reached a dead end. At that point, it would retrace its steps to the starting location and choose the next best path based on the best smell strategy. This path would then be followed until locating the cheese or finding another dead end. This cycle repeats until the cheese is found.

In the second method, the mouse follows the path based on the best smell strategy until a dead end is reached. Then it backs up only to the first previous step that has alternate movement choices. Here the next best choice is made and the process continues.

The second method was chosen because it more closely resembles human behavior when using the best smell strategy. The mouse having followed its nose to someplace near the cheese will do a local search before moving farther from its destination. This could prove very frustrating to a human since the cheese might very well be just over a wall. However, in general, a local search strategy is better.

3) Too much Initial Move Error

In the current system, if the physical motion of the mouse's move is too far off, the centering algorithm will not be able to locate the mouse. Therefore, where the mouse is and where the system thinks the mouse is are different. This is a hard failure to the system and the user must restart the traversal.

One strategy that could be used is the brute force search that is performed at start up. This search is lengthy and was found to be very rarely required.
5.3 Suggestions for Future Extensions

The following are only some of the possible extensions to this project. The test bed developed allows for many such extensions with a wide variety of depth.

1) Parallel Image Processing

The slowest component by far in the system was the image processing. Locating maze options in the vicinity of the mouse introduced the longest delay between moves. Moving this process to a computer or several others different from the PC could greatly speed up the process.

The nature of the image processing is such that it allows exploitation of parallelism. The single board controller that is currently used for motion control resides in a multislot Multibus card cage. Several of the National Controllers could be set up to run in parallel to process maze information. A simple approach would be to have one controller for each direction: North, South, East or West. These controllers would prove very useful as more sophisticated image processing algorithms and capabilities are developed. They would also serve to isolate the motion control entirely from the intelligent host. Even the closed loop feed back for the centering algorithm would be contained within the motion control hardware.

The major limitation for this expansion is the need to purchase or design a Multibus compatible frame grabber that can be connected to a CCD video camera. If design is chosen, this itself might prove to be a research project.

2) Optimum Search Strategy

The search strategy currently used only finds a solution to the maze problem. A great deal of work could be done in developing an optimum search path. This could be based on the best smell strategy or any new senses one might care to add to the mouse. This is a very complicated undertaking since no maze knowledge exists until a path is tried.

An undertaking not so ambitious might be to try several different search strategies. The performance of each optimized by execution time, number of steps, backtracks or other user selected metrics.
3) Infinite or "limited" Maze Corridor View

As mentioned in Section 5.2 - Remaining Limitations, the mouse is currently limited to one step look ahead. The development of more complicated image processing could provide a user selected depth of vision. The mouse could then be considered traversing a foggy maze. The thickness of this fog controlled by the user.

Additionally, the image processing may include views of the maze other than only North, South, East or West. By giving the mouse the ability to turn its head in increments smaller than 90 degrees, movement options that occur say two or three steps down the hall can be evaluated by the search algorithm.

This allows much greater flexibility in traversing the maze and would certainly be helpful in finding an optimum as well as more interesting maze paths.

4) Corrupt Knowledge

An area that was not explored in depth in this project was dealing with corrupt knowledge. This corrupt knowledge occurs because the maze can be altered during traversal. Currently the changes in maze geometry are noted and new movement options are update. However, using this new information to find a better path to the cheese is not done. Certainly, more sophisticated search strategies would have to exploit this information.

5) Moving Cheese

A variation that might prove interesting is to allow the cheese or target to move during traversal as well. Perhaps this can be thought of as having two mice searching for each other. They could have the same or different search strategies and senses. These results could then be compared to those of stationary targets.

Another twist is to have a mouse fleeing from a cat. In this case the one target is allowed to move, but now its objective is to never be found by the other.

These cases and the development of search strategies could be performed without a physical maze. The system is currently able to support software simulation for the development of these activities with visualization performed on the video monitor.
6) Better Physical Maze

Simply stated, any improvements to the physical maze hardware should be done. Major modifications would have to be done if there are multiple moving targets.

7) User Disable of Knowledge Display

A minor enhancement to the system is to allow the user to toggle the knowledge display. It currently takes longer than one might want. The display of the knowledge base after every move may not be necessary since the amount of knowledge gained is very little. By allowing the user to select how often, if ever, they wish the knowledge to be shown, the system's execution time could be shortened.
6.0 Appendices

Appendix A

Source Listing for TRAVERSE.BAT
del d:.*
del thesis.dba
copy startup.com d:
copy runtime.com d:
auto.com
Appendix B

Setting up the National DB32016
The following is the download procedure for loading the motion control software onto the National computer. It assumes that the necessary hardware configuration for communication with the vax cluster has already taken place. For more information on hardware configurations, see the DB32016 Development Board User's manual.

1) Depress the reset switch on the National board. The reset banner displayed on the screen indicates that communication with the monitor has been established.

   R VERSION _n.nn_dd-mm-yy

   Enter !<CR>. The board will respond with an '*' prompt. Then enter the monitor command omt <CR><CR>. You should now be connected to the vax.

2) Login to the vax.

3) Invoke the GNX Debugger

   $DBG16/noindirect
   ->sl tt:<CR>
   ->sr 4h<CR>
   ->b center.exe/sp=0fff0<CR>
   ->g

   At this point, the program is running. When you are done with maze traversal, use <CTRL>C to get back to the vax. Logout as usual.
Appendix C

Software Listing for COMMON.PAS
(* These routines are contained in a file named COMMON.PAS. They are the routines that are used in both the Startup and Runtime Image Processing Modules. *)

AUTHOR : ROBERT SPINA
DATE : 15 OCTOBER 1988

FUNCTION Get_Pixel(x,y:INTEGER):INTEGER;

(* This function returns the grey scale value of the pixel at location (x,y) in the Datacube Frame Grabber. The variable bank is required since the 512 X 512 pixels are addressed as 4 banks of 128 X 512. *)

VAR
    bank : BYTE;

BEGIN
    IF (x<=127) THEN bank := 16;
    IF ((x>128) AND (x<255)) THEN bank := 17;
    IF ((x>256) AND (x<383)) THEN bank := 18;
    IF (x>384) THEN bank := 19;

    hold(bank);
    x := x - ((bank-16)*128);
    Get_Pixel := datacube[x,y];
END;

PROCEDURE Pause;

(* This procedure is used to produce a delay between transmitting a movement selection to the real time controller and receiving a DONE acknowledge. *)

VAR
    i : INTEGER;
    c : CHAR;

BEGIN
    FOR i := 1 TO 10000 DO
        c := 'a';
END;

PROCEDURE Put_Pixel(x,y:INTEGER;v : BYTE);

(* This procedure will set the pixel located at (x,y) in the Datacube Frame Grabber to the value in v. The value of v can range from 0 to 255. White is 0 and Black is 255. *)

VAR
bank : BYTE;

BEGIN
  IF (x<=127) THEN bank := 16;
  IF ((x>=128) AND (x<=255)) THEN bank := 17;
  IF ((x>=256) AND (x<=383)) THEN bank := 18;
  IF (x>=384) THEN bank := 19;

  hold(bank);
  x := x - ((bank-16)*128);
  datacube[x,y] := v;
END;

FUNCTION Is_it_a_wall(x,y:INTEGER):boolean;

(* This function returns a boolean value that indicates whether the logical grid location given by (x,y) is a wall or an opening. A wall will return a TRUE value. The logical grid size translates to a physical window that is 15 rows by 8 columns. The material used to create a wall on the physical maze must lie between 92 and 164 of the grey scale. *)

VAR count,r,c,top,offset,i,i2 : INTEGER;
bank : BYTE;

BEGIN
  (* convert a logical grid location into the start of a physical one *)
  r := x * 16;
  c := y * 8;
  count := 0;

  FOR i2 := r TO r+15 DO
    FOR i := c TO (c+7) DO
      IF ((get_pixel(i2,i) > 92) AND (get_pixel(i2,i) < 164))
        THEN count := count + 1;

  (* The total number of wall colored pixels must be greater than 45 *)
  IF (count > 45)
    THEN Is_it_a_Wall := true
  ELSE
    Is_it_a_Wall := false;
END;

PROCEDURE Options(x,y:INTEGER);

(* This procedure generates the step options available to the North,
East, South and West. \((x,y)\) is the current Logical Grid location of the mouse. These options are stored in a file named OPTIONS.DAT on the virtual disk drive.

Currently the number of steps are limited to

\[
\begin{align*}
0 & \text{ - NONE} \\
1 & \text{ - One step}
\end{align*}
\]

```plaintext
TYPE
  file_type = text;

VAR
  steps : file_type;
  val : INTEGER;
  a : string[2];

BEGIN
  Assign(steps,'d:options.dat');
  Rewrite(steps);

  (* Check location to the North *)
  IF ((x-1) < 0)
    THEN val := 0
  ELSE
    IF (is_it_a_wall(x-1,y))
      THEN val := 0
    ELSE
      val := 1;

  writeln(val);
  str(val,a);
  writeln(steps,a);

  (* Check location to the East *)
  IF ((y+1) > 64)
    THEN val := 0
  ELSE
    IF (is_it_a_wall(x,y+1))
      THEN val := 0
    ELSE
      val := 1;

  writeln(val);
  str(val,a);
  writeln(steps,a);

  (* Check location to the South *)
  IF ((x+1) > 32)
    THEN val := 0
  ELSE
    IF (is_it_a_wall(x+1,y))
      THEN val := 0
    ELSE
      val := 1;
```

46
writeln(val);
str(val,a);
writeln(steps,a);

(* Check location to the West *)
IF ((y-1) < 0)
    THEN val := 0
ELSE
    IF (is_it_a_wall(x,y-1))
        THEN val := 0
    ELSE
        val := 1;

writeln(val);
str(val,a);
writeln(steps,a);
close(steps);

FUNCTION In_Range(value : INTEGER): Boolean;

(* This function returns a boolean value to indicate whether a value is within
the grey scale range for the mouse. *)
BEGIN
    IF ((value >= Thresh1) AND (value <= Thresh2))
        THEN In_range := true
    ELSE
        In_Range := False;
END;

PROCEDURE Get_Percent(VAR p1,p2,p3,p4:REAL; x,y : INTEGER);

(* This procedure calculates the number of mouse colored pixels in each of the
four quadrants of a logical grid location (x,y). *)
VAR
    r,c,count,try,i2,i : INTEGER;
    count1,count2,count3,count4 : REAL;
BEGIN
    count := 0;
    count1 := 0.0;
    count2 := 0.0;
    count3 := 0.0;
    count4 := 0.0;

    r := x * 16;
    c := y * 8;
count := 0;
(* Compute for upper left quadrant *)
FOR i2 := r TO (r+7) DO
  FOR i := c TO (c+3) DO
    IF In_Range(get_pixel(i2,i))
      THEN count1 := count1 + 1.0;

(* Compute for upper right quadrant *)
FOR i2 := r TO (r+7) DO
  FOR i := (c+4) TO (c+7) DO
    IF In_Range(get_pixel(i2,i))
      THEN count2 := count2 + 1.0;

(* Compute for lower left quadrant *)
FOR i2 := (r+8) TO (r+15) DO
  FOR i := c TO (c+3) DO
    IF In_Range(get_pixel(i2,i))
      THEN count3 := count3 + 1.0;

(* Compute for lower right quadrant *)
FOR i2 := (r+8) TO (r+15) DO
  FOR i := (c+4) TO (c+7) DO
    IF In_Range(get_pixel(i2,i))
      THEN count4 := count4 + 1.0;

(* Compute percentages *)
p1 := count1 / (count1+count2+count3+count4);
p2 := count2 / (count1+count2+count3+count4);
p3 := count3 / (count1+count2+count3+count4);
p4 := count4 / (count1+count2+count3+count4);
END;

PROCEDURE Center_mouse(x,y : INTEGER);
(* This procedure is a higher level routine that determines if the mouse is centered in a logical grid location at (x,y). It provides the closed loop feedback for the real time controller. It also issues the proper commands to the real time controller to place the mouse in the center. *)
VAR
  p1,p2,p3,p4 : REAL;
BEGIN
  Get_Percent(p1,p2,p3,p4,x,y);
  (* Too far West, repeat until satisfied *)
  WHILE ((p1+p3) > 0.65) DO
    BEGIN
      Serial link('e');
      Hold($10);
    END;
END;
Hold($30);
Get_Percents(p1,p2,p3,p4,x,y);
END;

(* Too far North, repeat until satisfied *)
WHILE ((p1+p2) > 0.65) DO
BEGIN
Serial link('s');
Hold($10);
Hold($30);
Get_Percents(p1,p2,p3,p4,x,y);
END;

(* Too far East, repeat until satisfied *)
WHILE ((p2+p4) > 0.65) DO
BEGIN
Serial link('w');
Hold($10);
Hold($30);
Get_Percents(p1,p2,p3,p4,x,y);
END;

(* Too far South, repeat until satisfied *)
WHILE ((p3+p4) > 0.65) DO
BEGIN
Serial link('n');
Hold($10);
Hold($30);
Get_Percents(p1,p2,p3,p4,x,y);
END;

PROCEDURE Template(VAR Flag:boolean; x,y: INTEGER);
VAR
r,c,count,try,i2,i : INTEGER;
BEGIN
  count := 0;
  r := x * 16;
  c := y * 8;
  count := 0;
  FOR i2 := r TO (r+15) DO
    FOR i := c TO (c+7) DO
      IF In_Range(get_pixel(i2,i))
        THEN count := count + 1;
      IF (count > 40)
        THEN
          flag := false
        ELSE
          flag := true;
  END;
PROCEDURE Find_Mouse;

(* This routine determines the logical
grid location of the mouse. The
results are placed on a file named
MOUSE.DAT and WAS.DAT. *)

TYPE
  file_type1 = text;
  file_type2 = file of INTEGER;

VAR
  position1 : file_type1;
  position2 : file_type2;
  r,c,try,count,upper_limit,lower_limit : INTEGER;
  Not_Found : boolean;
  a : string[2];

BEGIN
  r := 0;
  c := 0;

  upper_limit := 63;
  lower_limit := c;
  Not_Found := true;

  WHILE (Not_Found) DO
    BEGIN
      c := c+1;
      IF (c>upper_limit) THEN
        BEGIN
          c := lower_limit;
          r := r+1;
        END;
      Template(Not_Found,r,c);
    END;
  Assign(position2,'d:was.dat');
  rewrite(position2);
  write(position2,r);
  write(position2,c);
  close(position2);
  Assign(position1,'d:mouse.dat');
  rewrite(position1);
  str(r,a);
  writeln(position1,a);
  str(c,a);
  writeln(position1,a);
  close(position1);

  (* Once mouse location is found go
  get movement options *)
  Options(r,c);

  50
(* Display current knowledge of maze *)
   Draw_Knowledge;

END;

PROCEDURE Place_cheese(x,y:INTEGER);
(* This routine draws the cheese icon
   at logical grid location (x,y) into
   the Datacube Frame Grabber memory. *)

VAR
   r,c,top,offset,i : INTEGER;
   bank : BYTE;

BEGIN
   IF (x <= 8) THEN bank := 16;
   IF (x> 8) AND (x<=16) THEN bank := 17;
   IF (x>16) AND (x<=24) THEN bank := 18;
   IF (x>24) THEN bank := 19;
   x := x - ((x div 16)*16);
   offset := y * 8;
   top := x * 16;

   hold(bank);

   FOR i := (offset+0) TO (offset+0) DO
      datacube[0+top,1] := $ff;
   FOR i := (offset+0) TO (offset+1) DO
      datacube[1+top,1] := $ff;
   FOR i := (offset+0) TO (offset+2) DO
      datacube[2+top,1] := $ff;
   FOR i := (offset+0) TO (offset+3) DO
      datacube[3+top,1] := $ff;
   FOR i := (offset+0) TO (offset+4) DO
      datacube[4+top,1] := $ff;
   FOR i := (offset+0) TO (offset+5) DO
      datacube[5+top,1] := $ff;
   FOR i := (offset+0) TO (offset+6) DO
      datacube[6+top,1] := $ff;
   FOR i := (offset+0) TO (offset+7) DO
      datacube[7+top,1] := $ff;
   FOR i := (offset+0) TO (offset+7) DO
      datacube[8+top,1] := $ff;
   FOR i := (offset+0) TO (offset+6) DO
      datacube[9+top,1] := $ff;
   FOR i := (offset+0) TO (offset+5) DO
      datacube[10+top,1] := $ff;
   FOR i := (offset+0) TO (offset+4) DO
      datacube[11+top,1] := $ff;
   FOR i := (offset+0) TO (offset+3) DO
      datacube[12+top,1] := $ff;
   FOR i := (offset+0) TO (offset+2) DO
      datacube[13+top,1] := $ff;
   FOR i := (offset+0) TO (offset+1) DO
      datacube[14+top,1] := $ff;
   FOR i := (offset+0) TO (offset+0) DO

datacube[15+top,i] := $ff;

PROCEDURE Draw_Cheese;
(* This routine is used at Startup to allow the user to select a logical grid location for the cheese. The user will stay in this routine until they are satisfied with their selection. *)

TYPE
  file_type = text;

VAR
  offset,i,r,c,top : INTEGER;
  bank : BYTE;
  flag : boolean;
  a : string[2];
  cheese_file : file_type;

BEGIN
  writeln('This is to find place the cheese.');
  write('Enter the x offset: ');
  readln(offset);
  write('Enter the y offset: ');
  readln(top);
  place_cheese(offset,top);
  Assign(cheese_file,'d:cheese.dat');
  Rewrite(cheese_file);
  str(offset,a);
  writeln(cheese_file,a);
  str(top,a);
  writeln(cheese_file,a);
  close(cheese_file);
END;

PROCEDURE Draw_Grid;
(* This Procedure is used to display the logical grid boundaries on the video monitor. It is used to help the user align the physical maze with the logical maze. *)

VAR
  r,c : INTEGER;
  bank : BYTE;

BEGIN
  FOR bank := 16 TO 19 DO
  BEGIN
    
    52
hold(bank);
c := 0;
WHILE (c<512) DO
BEGIN
  FOR r := 0 TO 127 DO
  datacube[r,c] := $ff;
  c := c+8;
END;
END;
FOR bank := 16 TO 19 DO
BEGIN
  hold(bank);
r := 0;
WHILE (r<128) DO
BEGIN
  FOR c := 1 TO 511 DO
  datacube[r,c] := $ff;
  r := r+16;
END;
END;
END;
Appendix D

Software Listing for STARTUP.PAS
program startup;
{$R-J const
max_column = 363;
Thresh1 = 200;
Thresh2 = 255;
recv_buf_size = 2048;

{var
datacube : array[0..126, 0..511] of byte absolute $9000:0000;
screen : array[0..7, 0..7] of byte;
done : boolean;
mouse : Object_Type;
max_scanline : integer;

buf_start : buffer_pointer;
buf_end : buffer_pointer;
recv_buffer : array [1..recv_buf_size] of storage;
speed : integer;
dbBits : 7..8;
stop_bits : 1..2;
parity : check_bit;

[$I GRAPH.P]
[$I serial.pas]
[$I know.pas]
[$I common.pas]

procedure clear_pic;
/* This routine initializes the mouse's
knowledge of the maze. It will destroy
knowledge from a previous run and
create the file C:KNOW.PIC with initial
maze knowledge values of zero. */

var
filel : file of integer;
x,value : integer;

Begin
assign(filel,'c:know.pic');
 rewrite(filel);
value := 0;
for x := 1 to 2048 do
  write(file1,value);

close(file1);

procedure Set_Screen;
/* This routine is used to set the
  PC screen for display. */

Var
  i : integer;

Begin
  Port[$302] := $30;
  for i := 0 to 255 do
    Begin
      Port[$302] := 1;
      Port[$303] := 1;
      Port[$304] := 1;
      Port[$305] := 1;
      Port[$306] := 1;
    End;
  End;

procedure Start_up;
/* This routine does most of the initialization
  work for maze traversal. It will clear previous
  maze knowledge, query the user for cheese location
  and allow for user motion of the mouse in the
  physical maze. */

var
  c : char;
  flag : boolean;

Begin
  Set_Screen;
  flag := true;
  clear_pic;
  max_scanline := 127;
  while (flag) do
    begin
      Hold($10);
      Hold($30);
      Draw_cheese;
      c := 'w';
      /* This section allows the user to move the
         mouse in the physical maze. The user's
         options are N,S,E,W,n,s,e,w, or q to QUIT */
      while (c <> 'q') do
        begin
          write('enter command');
          readln(c);
        end
    end
End;
if (c <> 'q') then serial_link(c);
end;
Draw_Grid;
write('Are you satisfied ? ');
readln(c);
if (c = 'y') then flag := false;
end;

Hold($10);
Hold($30);
End;

{ start of main }
Begin
  Start_up;
  Find_Mouse;
  textBackground(black);
  textcolor(white);
  clrscr;
end.
Appendix E

Software Listing for RUNTIME.PAS
These routines are contained in a file named RUNTIME.PAS. They are the routines unique to maze traversal after initialization (STARTUP).

AUTHOR: ROBERT SPINA
DATE: 15 OCTOBER 1988

program runtime;
{$R-}
const
max_column = 363;
Thresh1 = 200;
Thresh2 = 255;
recv_buf_size = 2048;

type
  str255 = string[255];
  Object_type = Array[1..15,1..15] of byte;
  buffer_pointer = integer;
  smallstring = string[2];
  bigstring = string[255];
  storage = byte;
  check_bit = (none,even);

var
datacube : array[0..126, 0..511] of byte absolute $9000:0000;
screen : array[0..7, 0..7] of byte;
done : boolean;
mouse : Object_Type;
max_scanline : integer;
current_x, current_y, new_x, new_y : integer;
flag : Boolean;

buf_start : buffer_pointer;
buf_end : buffer_pointer;
recv_buffer : array[1..recv_buf_size] of storage;
speed : integer;
dbits : 7..8;
stop_bits : 1..2;
parity : check_bit;

{$I GRAPH.P}
{$I serial.pas}
{$I know.pas}
{$I common.pas}

procedure retrieve_mouse_location(Var x,y: integer);
/* This routine reads in the mouse's current location from the file D:WAS.DAT. This file contains the X,Y coordinate where the image processing software determined the mouse to be. */

type
  file_type = file of integer;
var
  position : file_type;

Begin
  Assign(position,'d:was.dat');
reset(position);
read(position,x);
read(position,y);
close(position);
end;

procedure save_mouse_location(Var x,y: integer);
/* This routine is used by the image processing
software to store the current mouse location
in the file D:WAS.DAT. */
type
   file_type = file of integer;
var
   position : file_type;
Begin
   Assign(position,'d:was.dat');
   rewrite(position);
   write(position,x);
   write(position,y);
   close(position);
end;

procedure tweak(new_x,new_y:integer; dir : char);
/* This routine is used after the initial move
of the mouse has been executed. It will continue
to issue small steps to the motion controller
in the selected direction until the mouse is
at least within the boundaries of the desired
logical grid location. */
var
   flag : boolean;
Begin
   Template(flag,new_x,New_y);
   while flag do
      begin
         serial_link(dir);
         Hold($10);
         Hold($30);
         Template(flag,new_x,new_y);
      end;
end;

procedure get_move(var x,y : integer);
/* This routine will read the file D:CHOICE.DAT
that was generated by the PROLOG search
algorithm. It contains the direction that the
mouse is to move. */
type
   file_type = text;
Var
a: string[1];
move: file_type;
code: integer;

Begin
Assign(move,'d:choice.dat');
reset(move);
readln(move,a);
close(move);

if (a[1] = 'n') then
begin
  serial_link('N');
  x := x - 1;
  tweak(x,y,'n');
end;

if (a[1] = 'e') then
begin
  serial_link('E');
  y := y + 1;
  tweak(x,y,'e');
end;

if (a[1] = 's') then
begin
  serial_link('S');
  x := x + 1;
  tweak(x,y,'s');
end;

if (a[1] = 'w') then
begin
  serial_link('W');
  y := y - 1;
  tweak(x,y,'w');
end;

End;

procedure Draw_Cheese_second;
/* This routine refreshes the cheese icon on the video monitor. */
type
  file_type = text;
var
  code,offset,i,r,c,top: integer;
  bank: byte;
  flag: boolean;
  a: string[2];
  cheese_file: file_type;
begin
  Assign(cheese_file,'d:cheese.dat');
  Reset(cheese_file);
readln(cheese_file,a);
Val(a,offset,code);
readln(cheese_file,a);
Val(a,top,code);
close(cheese_file);

place_cheese(offset,top);

end;

{Start of MAIN}
Begin
max_scanline := 127;
Hold($10);
Hold($30);

retrieve_mouse_location(current_x,current_y);

new_x := current_x;
new_y := current_y;

Get_Move(new_x,new_y);

Hold($10);
Hold($30);

Pause;
Center_Mouse(new_x,new_y);

save_mouse_location(new_x,new_y);
Option(new_x,new_y);

Draw_Cheese_Second;
textBackground(black);
textcolor(white);
clrscr;

Draw_Knowledge;
end.
Appendix F

Software Listing for SERIAL.PAS
Description:

Communications routines for TURBO Pascal written by Alan Bishop
Handles standard COM1 ports with interrupt handling. Includes
support for only one port, and with no overflow, parity, or other
such checking. However, even some of the best communication programs
don't do this anyway.

Procedure Check_Range (var range : integer);

{This is used to adjust buffer pointers.}
Begin
  if range > recv_buf_size then range := 1;
End;

Function Compressed : boolean;

{Like keypressed, but for the comm port.}
{Same function partly as Borland's Async_Buffer_Check routine.}
Begin
  Compressed := (buf_start <> buf_end);
End;

Function Cinkey : char;

{Returns nothing or a code from the buffer - 2 bytes are used for
ease of use with a two byte inkey routine.}
{Same function partly as Borland's Async_Buffer_Check routine.}
Var
  result : char;
  temp : integer;
Begin
  if not Compressed then result := ' ';
  else
    begin
      inline ($FA);
      temp := recv_buffer[buf_start];
      buf_start := buf_start + 1;
      Check_Range (buf_start);
      inline ($FB);
      result := Chr (temp);
      End;
      Cinkey := result;
    End;

Function Carrier : boolean;

{True if carrier, false if not}
Begin
  Carrier := odd(port[$3FE] shr 7);
End;

Procedure Set_Up_Recv_Buffer;

{Big procedure isn’t it?}
{Crucial for initializing to empty the 2k comm port buffer.}

Begin
  buf_start := 1;
  buf_end := 1;
End;

Procedure Set_Baud (rate : integer);

{No problems with non-standard bauds.}
{Same function as Borland’s Async_Change routine.}

Var
  a : byte;
  divided : real;

Begin
  if rate <= 9600 then
    begin
      speed := rate;
      divided := 115200.0/rate;
      rate := trunc(divided);
      a := port[$3fb];
      if a < 128 then a := a+128;
      port[$3fb] := a;
      port[$3f8] := lo(rate);
      port[$3f9] := hi(rate);
      port[$3fb] := a-128;
    end;
End;

Procedure Update_Uart;

{Uses dbits, stop_bits, and parity.}

Var a : byte;

Begin
  a := dbits-5;
  if stop_bits = 2 then a := a + 4;
  if parity = even then a := a + 24;
  port[$3fb] := a;
End;

Procedure Init_Port;

{Sets up most anything necessary.}
{Same function as Borland’s Async_Isr routine.}

Var
  a, b : integer;
  buf_len : integer;
Begin
Update Uart;
port[$3f9] := 1; \{interrupt enable\}
a := port[$3fc];
if odd(a) then a := 1 else a := 0; \{keep terminal rea\}
a := a + 10;
port[$3fc] := a; \{turn on req to se\}
a := port[$3fa];
port[$21] := $c;
Set_Baud (speed);
buf_len := recv_buf_size;

{*** This is the background routine. ***}
inline {
$1E/
$0E/
$1F/
$BA/+23/
$B8/$0C/$25/
$CD/$21/
$8B/$BE/BUF_LEN/
$99/$3E/+87/
$1F/
$2E/$8C/$1E/+83/
$EB/$51/
$FB/
$1E/
$50/
$53/
$56/
$2E/$8E/$1E/+70/
$BA/$FB/$03/
$EC/
$BE/RECV_BUFFER/
$8B/$1E/BUF_END/
$88/$40/$FF/
$43/
$E8/$22/$00/
$89/$1E/BUF_END/
$3B/$1E/BUF_START/
$75/$0C/
$8B/$1E/BUF_START/
$43/
$E8/$10/$00/
$89/$1E/BUF_START/
$BA/$20/$00/
$BO/$20/
$EE/
$5E/
$5A/
$5B/
$58/
$1F/
$CF/
$2E/$8B/$16/+11/
$42/
$39/$DA/

66
Procedure Term_Ready (state : boolean);

[Send a true for on, false for off.]
[Same function partly as Borland's Async_Open and Async_Clos]

Var a : byte;

Begin
  a := port[$3fc];
  if odd(a) then a := a - 1;
  a := a + ord(state);
  port[$3fc] := a;
End;

Procedure Remove_Port;

[Gets rid of most problems.]
[Same function partly as Borland's Async_Close routine.]

Var a : byte;

Begin
  port[$3f9] := 0;
  a := port[$3fc];
  if odd(a) then a := 1 else a := 0;
  port[$3fc] := a;
  port[$21] := $BC;
End;

Procedure Write_Byte (to_send : bigstring);

{Sends out up to 255 bytes.}
{Same function as Borland's Async_Send routine.}

Var a, b, c : byte;

Begin
  for b := 1 to length (to_send) do begin
    c := ord (to_send[b]);
    repeat
      a := port[$3fd];
      until odd(a shr 5);
    port[$3f8] := c;
  end;
End;

Procedure Break;

{Gimme a break.}
Var a, b : byte;

Begin
    a := port[$3fb];
    b := a;
    if b > 127 then b := b - 128;
    if b <= 63 then b := b + 64;
    port[$3fb] := b;
    delay (400);
    port[$3fb] := a;
End;

Procedure Setup;
{Initialize most stuff – you may want to replace this routine
  {Same function partly as Borland's Async_Open routine.}

Var a : byte;

Begin
    dbits := 8;
    parity := none;
    stop_bits := 1;
    speed := 4800;
    Init_Port;
    Term_Ready (true);
End;

(*-----------------------------------------------
Procedure Serial_link(direction:char);

{ Don’t allow <cntl> C because the port will not be removed
  or the terminal returned. This will cause problems for Turbo.

{ C- }

Const
    leave : boolean = false;
    reinit_buffer : boolean = true;

Var
    key : char;
    bert_char : char;
    bert_string : string[80];
    real_string : string[10];
    real_number : real;
    error_code : integer;
    window_count : integer;
    xmin, xmax : integer;
    ymin, ymax : real;
    x_position, y_position : integer;
    i, j : integer;

Begin
    Setup;

   68
Write_Byte (direction + 'M);

(Some delay is needed before initializing the comm port buffer so that sending the commands above does not erroneously put in the buffer after this next call to set_up_recv_buffer has initialized that buffer. (It's some kind of delayed reaction.)

xmin := 1;
xmax := 300;
ymin := 32000;
ymax := 0;

window_count := 0;
key := ' ';
leave := false;

Set Up Recv Buffer;    (This will initialize the communica

repeat
  if Compressed then
    begin
      bert_char := Cinkey;
      bert_string := concat (bert_string, bert_char);
    end;
  until (bert_char = 'd');

END;
Appendix G

Software Listing for KNOW.PAS
These routines are contained in a file named KNOW.PAS. They are the routines used to display the mouse’s knowledge of the maze on the PC screen.

AUTHOR: ROBERT SPINA
DATE: 15 OCTOBER 1988

procedure Draw_Knowledge;
const
deltax = 5;
deltay = 6;
full: array[0..7] of byte = ($ff,$ff,$ff,$ff,$ff,$ff,$ff,$ff
half: array[0..7] of byte = ($55,$aa,$55,$aa,$55,$aa,$55,$aa

var
mouse, y_mouse, x, y, xgrid, ygrid, color : integer;
block: array[0..63,0..31] of integer;

procedure init;
/* Set up the screen in graphics mode */
begin
  graphcolormode;
  graphbackground(0);
  palette(0);
  pattern(full);
end;

procedure get_pic;
/* Bring in the mouse's current knowledge of the maze from the file C:KNOW.PIC. */

var
  file1 : file of integer;

begin
  assign(file1,'c:know.pic');
  reset(file1);
  for ygrid:=0 to 31 do
    for xgrid:=0 to 63 do
      read(file1,block[xgrid,ygrid]);
  close(file1);
end;

procedure choose_color(x,y:integer; ch,d : char);
/* This routine will select the proper color to be displayed for a maze location. The color depends on whether there is corrupt knowledge or new knowledge. */

Begin

71
if (d = 'n') then x := x - 1;
if (d = 'e') then y := y + 1;
if (d = 's') then x := x + 1;
if (d = 'w') then y := y - 1;

if (ch = '0') and (block[y, x] = 0) then block[y, x] := 0;
if (ch = '1') and (block[y, x] = 0) then block[y, x] := 1;
if (ch = '0') and (block[y, x] = 1) then block[y, x] := 2;
if (ch = '1') and (block[y, x] = 1) then block[y, x] := 1;
if (ch = '0') and (block[y, x] = 2) then block[y, x] := 2;
if (ch = '1') and (block[y, x] = 2) then block[y, x] := 3;
if (ch = '0') and (block[y, x] = 3) then block[y, x] := 2;
if (ch = '1') and (block[y, x] = 3) then block[y, x] := 3;

End;

procedure update_pic;
/* This routine updates the mouse's knowledge
   based on the new move that occurred. */
var
   file1 : file of integer;
   file2 : text;
   c : string[1];

Begin
  assign(file1,'d:was.dat');
  reset(file1);
  read(file1,x_mouse);
  read(file1,Y_Mouse);
  close(file1);

  assign(file2,'d:options.dat');
  reset(file2);
  readln(file2,c);
  choose_color(x_mouse,y_mouse,c,'n');
  readln(file2,c);
  choose_color(x_mouse,y_mouse,c,'e');
  readln(file2,c);
  choose_color(x_mouse,y_mouse,c,'s');
  readln(file2,c);
  choose_color(x_mouse,y_mouse,c,'w');
  close(file2);

End;

procedure save_pic;
/* This routine saves the updated maze knowledge
for subsequent steps. */

```pascal
var
  file1 : file of integer;
begin
  assign(file1,'know.pic');
  rewrite(file1);
  for ygrid:=0 to 31 do
    for xgrid:=0 to 63 do
      write(file1,block[xgrid,ygrid]);
  close(file1);
end;
```

```pascal
procedure fillmaze;
  /* This routine will actually display the proper color for each of the grid locations based on the knowledge of the mouse. */
begin
  for ygrid:=0 to 31 do
    for xgrid:=0 to 63 do
      begin
        x := xgrid*deltax;
        y := ygrid*deltay;
        if (block[xgrid,ygrid] <= 3) then
          begin
            color := block[xgrid,ygrid];
            pattern(full);
          end
        else
          pattern(half);
        case (block[xgrid,ygrid]) of
          4: color := 1;
          5: color := 2;
          6: color := 3;
        end;
        fillpattern(x,y,x+deltax,y+deltay,color);
      end;
end;
```

```pascal
procedure writeblock(xgrid,ygrid,color:integer);
begin
  if (color > 6) then color:=0;
  block[xgrid,ygrid]:= color;
  fillmaze;
end;
```
begin (main)
get_pic;
update_pic;
init;
block(y_mouse,x_mouse) := block(y_mouse,x_mouse) + 3;
fillmaze;
textmode(c80);
block(y_mouse,x_mouse) := block(y_mouse,x_mouse) - 3;
save_pic;
end;
Appendix H

Software Listing for AUTO.PRO
domains
dn = integer
file = talking_pascal
integerlist = Integer*
database
current_mouse(integer, integer)
cheese(Integer, integer)
have Been(integer, integer)
options_at(integer, integer, symbol, integer, symbol)
how_i_got_to(integer, integer, symbol, integer)
predicates
where_is_cheese
where_is_mouse
g et_options
move(integer, integer, symbol, integer, integer, integer)
found_cheese
find_cheese
where_to_move(integerlist)
back_up
step_back(symbol, integer, integer, integer)
update_location(integer, integer, integer, integer, symbol, integer)
update_options(integer, integer, symbol, integer)
have_iBeen(integer, integer, symbol, integer)
g et_smell(integerlist, integerlist)
g et_smells(integerlist, integerlist)
g et_smelle(integerlist, integerlist)
g et_smellw(integerlist, integerlist)
w rite_list(integerlist)
b est_smell(integer, integerlist)
goal
system("d:init.com")
where_is_cheese,
where_is_mouse,
current_mouse(X,Y),
assertaThow_i_got_to(X,Y,"n",0)),
find_cheese.
clauses
write_list([]).
write_list([Head|Tail]):-write(Head),nl,write_list(Tail).
/* This only compensates for one step. If you need more steps
you would have to use the information in options_at(). */
g et_smellw(S1,S2):-
cheese(Cx,Cy),current_mouse(Mx,My),
Swx = abs(Mx-Cx) * abs(Mx-Cx),
Swy = abs(My-1-Cy) * abs(My-1-Cy),
Smellw = Swx+Swy,options_at(Mx,My,"w",D,"u"),D>0,
write("WEST = "),write(Smellw),nl,
get_smellw(S1,S2) :-
    S2 = S1.

get_smells(S1,S2) :-
    cheese(Cx,Cy),
current_mouse(Mx,My),
    Ssx = abs(Mx+1-Cx) * abs(Mx+1-Cx),
    Ssy = abs(My-Cy) * abs(My-Cy),
    Smells = Ssx+Ssy,
    options at(Mx,My,"s",D,"u"), D>0,
    write("SOUTH = "), write(Smells), nl,
    S2 = [Smells|S1].

get_smells(S1,S2) :-
    S2 = S1.

get_smelln(S1,S2) :-
    cheese(Cx,Cy),
current_mouse(Mx,My),
    Snx = abs(Mx-1-Cx) * abs(Mx-1-Cx),
    Sny = abs(My-Cy) * abs(My-Cy),
    Smelln = Snx+Sny,
    options at(Mx,My,"n",D,"u"), D>0,
    write("NORTH = "), write(Smelln), nl,
    S2 = [Smelln|S1].

get_smelln(S1,S2) :-
    S2 = S1.

best_smell(_,[]).

best_smell(X,[Head|Tail]) :-
    X <= Head,
    best_smell(X,Tail).

where_is_cheese :-
    openread(talking_pascal,"d:cheese.dat"),
    readdevice(talking_pascal),
    readln(X_string),
    str int(X_string,X_cheese),
    readln(Y_string),
    str int(Y_string,Y_cheese),
    readdevice(keyboard),
    closefile(talking_pascal),
    write("X of cheese is "), write(X_cheese), nl,
    write("Y of cheese is "), write(Y_cheese),
    asserta(cheese(X_cheese,Y_cheese)), nl.

where_is_mouse :-
    openread(talking_pascal,"d:mouse.dat"),
    readdevice(talking_pascal),
str_int(X_string,X_MOUSE),
readln(Y_string),
str_int(Y_string,Y_MOUSE),
readdevice(keyboard),
closefile(talking_pascal),
write("X of mouse is "),write(X_MOUSE),nl,
write("Y of mouse is "),write(Y_MOUSE),nl,
asserta(current_mouse(X_MOUSE,Y_MOUSE)),
asserta(have Been(X_MOUSE,Y_MOUSE)).

have_i_been(X,Y,Dir,DIST) :-
not(options_at(X,Y,Dir,DIST,"t")),
asserta(options_at(X,Y,Dir,DIST,"u")).

have_i_been(X,Y,Dir,DIST) :-
options_at(X,Y,Dir,DIST,"t").

update_OPTIONS(X,Y,Dir,DIST) :-
not(options_at(X,Y,Dir,DIST,_)),
asserta(options_at(X,Y,Dir,DIST,"u").

get_options :-
openread(talking_pascal,"d:options.dat"),
readdevice(talking_pascal),
readln(N),str_int(N,Dn),current_mouse(X,Y),
update_Options(X,Y,"n",Dn),
readln(E),str_int(E,De),current_mouse(X,Y),
update_Options(X,Y,"e",De),
readln(S),str_int(S,Ds),current_mouse(X,Y),
update_Options(X,Y,"s",Ds),
readln(W),str_int(W,Dw),current_mouse(X,Y),
update_Options(X,Y,"w",Dw),
closefile(talking_pascal).

update_location(Old_x,Old_y,New_x,New_y,Dir,DIST) :-
retract(current_mouse(_,_,_)),
asserta(current_mouse(New_x,New_y)),
retract(options_at(Old_x,Old_y,Dir,_,_)),
asserta(options_at(New_x,New_y,Dir,_,_)),
asserta(how_i_got_to(New_x,New_y,Dir,DIST)),
write(" I moved ",write(Dir),write(" My new position "),write(New_x),write(" "),write(New_y),write(" "),nl.

move(Old_x,Old_y,Dir,DIST,Delta_x,Delta_y) :-
DIST > 0,
New_x = (Old_x + Delta_x), New_y = (Old_y + Delta_y),
not(options_at(New_x,New_y,_,_,_) ),
update_location(Old_x,Old_y,New_x,New_y,Dir,DIST),
openwrite(talking_pascal,"d:choice.data"),
writedevice(talking_pascal),
write(Dir),nl,
closefile(talking_pascal),writedevice(screen).

step_back("n",X,Y,DISTANCE) :-
New_x = (X+DISTANCE), New_y = Y,
asserta(current_mouse(New_x,New_y)),
write(" I backed up. My new position is ").
write(New_x), write("\"\"), write(New_y), write("\")\"), nl,
openwrite(talking_pascal, "d:choice.dat"),
write\ devices\ (talking_pascal),
write('s'), nl,
closefile(talking_pascal), write\ device\ (screen).

step_back("e", X, Y, Distance) :-
New_x = X, New_y = (Y - Distance),
asserta(current_mouse(New_x, New_y)),
write("I backed up. My new position is "),
write\ (New_x), write("\"), write\ (New_y), write\ ("\")\"), nl,
openwrite\ (talking_pascal, "d:choice.dat"),
write\ device\ (talking_pascal),
write('e'), nl,
closefile\ (talking_pascal), write\ device\ (screen).

step_back("s", X, Y, Distance) :-
New_x = (X - Distance), New_y = Y,
asserta\ (current\ mouse\ (New_x, New_y)),
write("I backed up. My new position is "),
write\ (New_x), write\ ("\"), write\ (New_y), write\ ("\")\"), nl,
openwrite\ (talking_pascal, "d:choice.dat"),
write\ device\ (talking_pascal),
write('s'), nl,
closefile\ (talking_pascal), write\ device\ (screen).

step_back("w", X, Y, Distance) :-
New_x = X, New_y = (Y + Distance),
asserta\ (current\ mouse\ (New_x, New_y)),
write("I backed up. My new position is "),
write\ (New_x), write\ ("\"), write\ (New_y), write\ ("\")\"), nl,
openwrite\ (talking_pascal, "d:choice.dat"),
write\ device\ (talking_pascal),
write('w'), nl,
closefile\ (talking_pascal), write\ device\ (screen).

back_up :-
current\ mouse\ (X, Y),
retract\ (current\ mouse\ (\ , \ )),
how\ i\ got\ to\ (X, Y, Dir, Dist),
retract\ (how\ i\ got\ to\ (X, Y, Dir, Dist)),
step\ back\ (Dir, X, Y, Dist).

where\ to\ move\ (Smell) :-
current\ mouse\ (Mx, My), options\ at\ (X, Y, "n", D, "u"), D > 0,
write\ list\ (Smell),
Dx = (-1*D), Dy = 0, cheese\ (Cx, Cy),
Snx = abs\ (Mx - 1 - Cx) * abs\ (Mx - 1 - Cx),
Sny = abs\ (My - Cy) * abs\ (My - Cy),
Smelln = Snx + Sny, best\ smell\ (Smelln, Smell),
write("North is best"), nl,
move\ (X, Y, "n", D, Dx, Dy);

current\ mouse\ (Mx, My), options\ at\ (X, Y, "e", D, "u"), D > 0,
write\ list\ (Smell),
Dx = 0, Dy = D, cheese\ (Cx, Cy),
Sex = abs\ (Mx - Cx) * abs\ (Mx - Cx),
Sey = abs\ (My - 1 - Cy) * abs\ (My - 1 - Cy),
Smelle = Sex + Sey, best\ smell\ (Smelle, Smell),
write("East is best”), nl,
move(X,Y,"e",D,Dx,Dy);

current_mouse(Mx,My),options_at(X,Y,"s",D,"u"),D>0,
write(Smell),
Dx = D,Dy=0,cheese(Cx,Cy),
Ssx = abs(Mx+1-Cx) * abs(Mx+1-Cx),
Ssy = abs(My-Cy) * abs(My-Cy),
Smells = Ssx+Ssy,best_smell(Smells,Smell),
write("South is best"),nl,
move(X,Y,"s",D,Dx,Dy);

current_mouse(Mx,My),options_at(X,Y,"w",D,"u"),D>0,
write(Smell),
Dx = 0,Dy=(-1*D),cheese(Cx,Cy),
Swx = abs(Mx-Cx) * abs(Mx-Cx),
SWy = abs(My-1-Cy) * abs(My-1-Cy),
Smellw = Swx+Swy,best_smell(Smellw,Smell),
write("West is best"),nl,
move(X,Y,"w",D,Dx,Dy);

back_up.

found_cheese :-
current_mouse(X,Y),
options_at(X,Y,"n",Dist, ),
New_x = (X - Dist), New_y = Y,
cheese(New_x,New_y).

found_cheese :-
current_mouse(X,Y),
options_at(X,Y,"e",Dist, ),
New_x = X, New_y = (Y + Dist),
cheese(New_x,New_y).

found_cheese :-
current_mouse(X,Y),
options_at(X,Y,"s",Dist, ),
New_x = (X + Dist), New_y = Y,
cheese(New_x,New_y).

found_cheese :-
current_mouse(X,Y),
options_at(X,Y,"w",Dist, ),
New_x = X, New_y = (Y - Dist),
cheese(New_x,New_y).

find_cheese :-
get_options,
get_smell([],Smell1),
get_smells(Smell1,Smell2),
get_smell(Smell2,Smell3),
get_smellw(Smell3,Smell),
not(found_cheese),
where_to_move(Smell),
system("d:runtime.com"),
find_cheese.

find_cheese :-
write("I found the cheese!!!"),
save("thesis.db3").
Appendix I

Software Listing for CENTER.C
/* This program is contained in a file named CENTER.C and must be located on the vax cluster. It must be compiled with the GNX C Optimizing Compiler.

AUTHOR: ROBERT SPINA
DATE: 15 OCTOBER 1988 */

main()
{
char keystroke;
char previous;
unsigned char *ctrl_8255;

/* This defines the physical address of the 8255 on the National card and initializes it to be MODE 0 with all ports OUTPUT. */

ctrl_8255 = (unsigned char *)0x0c00026;
*ctrl_8255 = 0x80;
previous = ' ';

/* Here we monitor the PC requests for movement. as long as we do not receive a q (for QUIT) we process the motion request. */

while ((keystroke = getchar()) != 'q')
{
    if (keystroke == 'S') move_n_e(0xf0,10);
    if (keystroke == 'E') move_n_e(0x0f,5);
    if (keystroke == 'N') move_s_w(0xf0,10);
    if (keystroke == 'W') move_s_w(0x0f,5);

    if (keystroke == 's') move_n_e(0xf0,2);
    if (keystroke == 'e') move_n_e(0x0f,2);
    if (keystroke == 'n') move_s_w(0xf0,2);
    if (keystroke == 'w') move_s_w(0x0f,2);
}

move_n_e(mask,steps)
/* This routine will move the mouse either North or East the specified number of steps */

unsigned char mask;
int steps;
{
    int i;
    int count;
    unsigned char *porta;

    porta = (unsigned char *)0x0c00020;
for (count=1; count<steps; count++) {
    *porta = 0xaa | mask;
    /* Allow for inertia of stepper, by delaying 1000 cycles. */
    for (i=1; i<1000; i++) {}
    *porta = 0x99 | mask;
    for (i=1; i<1000; i++) {}
    *porta = 0x55 | mask;
    for (i=1; i<1000; i++) {}
    *porta = 0x66 | mask;
    for (i=1; i<1000; i++) {}
}

/* Free stepper motor to prevent overheating. */
*porta = 0xff;

/* Issue a done signal back to the PC */
printf("d");
}

move_s_w(mask, steps)
/* This routine will move the mouse either South or West the specified number of steps */

unsigned char mask;
int steps;
{
    int i;
    int count;
    unsigned char *porta;

    porta = (unsigned char *)0x0c00020;

    for (count=1; count<steps; count++) {
        *porta = 0xaa | mask;
        /* Allow for inertia of stepper, by delaying 1000 cycles. */
        for (i=1; i<1000; i++) {}
        *porta = 0x66 | mask;
        for (i=1; i<1000; i++) {}
        *porta = 0x55 | mask;
        for (i=1; i<1000; i++) {}
        *porta = 0x99 | mask;
        for (i=1; i<1000; i++) {}
    }

    /* Free stepper motor to prevent overheating. */
    *porta = 0xff;
}
/* Issue a done signal back to the PC */
printf("d");
7.0 References


FOOTNOTES
