An Interactive Environment to Investigate Robot Path Planning in a 2D Work-space

by

Khanh N. Hoang

B.S., University of Lowell, 1983
M.S., University of Lowell, 1986

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1989

Approved by:

Major Professor

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

1.1 Current Robot Programming Technology

Current Robot Programming Technology has become more and more sophisticated to satisfy the need for intelligent factory automation controllers in Computer Integrated Manufacturing. Industrial robots are essentially positioning devices. However, many robot systems today are better described as computer controlled manipulators. As more intelligence is required on the factory floor, these robot systems function as work cell controllers in networks of factory control systems.

A modern robot controller typically has the same basic components as a general purpose computer (Figure 1-1): A central processing unit (CPU), a memory subsystem, a mass-storage subsystem and a user interface. The additional components are a manipulator control unit, a control panel and teach pendant, a process control input/output interface, a network interface and possibly a machine vision subsystem. The manipulator control unit is usually made up of servo controllers and amplifiers that allow the CPU to drive the motors in the robot arm. A teach pendant is a hand-held switch and display box with which the robot arm can be controlled manually. A process control interface is typically made up of digital input/output lines primarily to synchronize the robot task with other devices such as conveyor motors, sensors, etc. Robot work cells are often built around a robot by integrating the process control directly into the robot controllers. Robot controllers communicate with each other and with other computers via their network interfaces. Vision systems are most commonly used in robot guidance and inspection. However not all robots have vision capability because vision systems often cost as much as robots. Therefore, unless it is really necessary, "blind" robots are better justified.

On the software side, robot control operating systems and high level programming languages provide a fairly high degree of flexibility. A few robot programming languages are modified versions of BASIC. Some others, such as VAL-II[1] are structured and modular. These robot languages are very similar to other programming languages. Their versatility provides the basic tool to build up factory automation

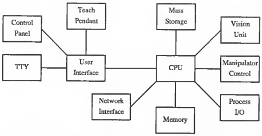

Figure 1-1. Block diagram of a modern robot controller

intelligence. In addition, they have a set of special commands and instructions tailored to the motion control task including a number of mathematical functions.

However, since the most common robot programming technique, "program by showing"[2], is not adaptive to configuration changes in the environment, better algorithms are needed to build more autonomous robots. "Program by showing" is the practice in which the robot arm is manually guided through specific motions and points are recorded for future repetition. This seems to be the most effective way to program robots used for spray painting or welding since most points on the trajectories are critical for such applications. In assembly processes, only pick-and-place points are the critical points, yet all points along the trajectories between them are explicitly "taught" to avoid obstacles in the work space. When the tasks change or when a work cell is duplicated with modifications, all these trajectories must be re-programmed. It seems to be unnecessary and wasteful when many non-critical points have to be specified over and over again.

A solution to this problem is to let the robot choose its own paths based on a knowledge of the work space. The question is how to inform the robot enough about its surroundings so that we can subsequently tell it to move from one point to another within its limits while avoiding all obstacles.

1.2 Algorithmic Motion Planning

Motion Planning is a rich mathematical field whose recent advances may become valuable contributions to the next generation of robots. Algorithmic motion planning involves the design and analysis of non-heuristic algorithms that are exact and asymptotically efficient in the worst case. Heuristic motion planning consists of the AI approaches that favor approximating, rule-based or best-case-tailored solutions. These approaches have proven to be successful in many situations. In a recent article, Micha Sharir[3] suggested that since the problem has a rich geometric and combinatorial structure, this structure should be understood from a mathematically rigorous point of view and algorithmic solutions should be sought first. Heuristic shortcuts would be helpful in complex cases where exact solutions might be computationally intractable.

General techniques for solving the motion planning problem have been found. Schwartz and Sharir[4] proved that this problem can be solved in time polynomial in the number n of algebraic geometric constraints defining the free configuration space but doubly exponential in k, the number of degrees of freedom of the robot. Canny[5] recently extended and improved this result to provide a solution in time $O(n^k \log n)$.

With the general algorithms above, the problem becomes intractable when the number of degrees of freedom k is large. However, when k is small these algorithms can solve the problem efficiently in time polynomial in the number of constraints n.

More recent researches have been aimed to improve algorithms for systems with a just a few degrees of freedom. The *projection* method is one in which the k-dimensional configuration space FP of the system B is decomposed into its pathwise connected components and the two positions of B, $P_{initial}$ and P_{final}, are to be determined whether they are in the same connected component of FP. This decomposition is done by projecting FP on to a sub-space A of lower dimension and then partitioning A into connected regions R.

The projection method has been applied by Schwartz and Sharir in the papers on the "piano movers" problem. Initial solutions were coarse and not very efficient (running time of $O(n^5)$). Using a modified projection technique, Leven and Sharir[6] designed a fairly efficient algorithm which runs in time $O(n^2 \log n)$. This consists of constructing cells and establishing adjacency in *FP*.

Other techniques subsequent to the *projection* technique have been considered, among them, the *retraction* approach. In the *retraction* method, the configuration space is further reduced to one-dimensional. The motion planning problem then becomes the *graph searching* problem[7]. O'Dunlaing and Yap[8] have applied this retraction method in the case of a disk moving in 2D polygonal space. This is made possible by constructing the *Vonoroi diagram*, which is defined as the subset of the configuration space *FP* of *B* consisting of placements of *B* simultaneously nearest to two or more obstacles. The Vonoroi diagram of *n* line segments in the plane can be computed in time $O(n \log n)$.

Another general technique, the *expanded obstacles* approach, has been playing an important role in many motion planning researches. Details of this technique will be explained later in this paper.

A variant of the motion planning problem deals with optimal paths. This is aimed to calculate the Euclidean shortest path between initial and final placements avoiding all obstacles. While work done on the 2D case have been successful, the 3D case is so complex that the problem becomes intractable.

In general, different techniques have been developed for the motion planning problem. However, as Sharir has indicated, although general algorithms are significant from a theoretical point of view, they are hopelessly inefficient in the worst case and are completely useless in practice.

1.3 A Practical Application

A step towards applying computational geometry in practical use is to model the physical environment in the system and to formulate efficient motion planning algorithms to help the robot navigate in its work envelope in a more autonomous manner.

This kind of improvement could be seen at two levels: Design and Application. At the design level, these algorithms are built into the programming language as instructions and commands or as part of the standard robot control system. Commands to describe the environment will be executed to set system parameters that will define the free configuration space. Innovation at the design level will take a long time to appear because of the usual long cycle between design conception, new product realization and marketing.

At the application level, motion planning algorithms can also be applied as part of application programs. The programmer is to store coordinates of the boundary points of the robot work space and around obstacles in the system. Based on that information, algorithmic motion planning programs can be written to make sure obstacles are avoided. Naturally, improvements at the application level are much more feasible since they do not necessarily require hardware changes.

In the rest of this paper we will limit our attention to algorithms at the application level. Chapter 2 suggests a two-dimensional model of robot work-space. Chapter 3 describes an algorithm that provides a simple solution to the robot path planning problem at the application level. Chapter 4 describes the simulation program that allows the integration of different algorithms in an interactive environment based on the model.

1.4 Realistic constraints

Realistic constraints concerning memory use and computation overhead incurred by the additional computation is worth serious considerations. Although most robot systems contain the basic components of general purpose computers, their resources such as processing time and especially memory and mass storage space are usually more limited. Thus, in developing these algorithms two issues are of concern: First, sophisticated motion planning algorithms added to regular applications will certainly be of value but they will undoubtedly require additional memory space. If they use too much memory, regular applications may suffer, or worse yet, may not be able to run at all. Second, these algorithms must be efficient to avoid performance degradation of the general task. If the robot is to compute the path from one point to another in its envelope without collision, it must be able to do it in a reasonably short time so that there is no apparent delay between command execution and actual robot motions. Otherwise, the additional overhead is not justified. In short, our goal here is to develop better path planning algorithms, but they must be simple and efficient in order to be practical.

2. A TWO-DIMENSIONAL MODEL

The following is a description of an interactive environment to facilitate the investigation and development of simple and practical algorithms to find collision-free paths between two points among obstacles in a 2D space. A model representing the work-space and the robot is necessary to serve as the foundation for all algorithms.

2.1 Basics of robot configurations

A robot arm, or manipulator, is basically a mechanical system of rigid links attached to each other at certain joints. The number of joints dictates the number of degrees of freedom of the arm. Typically, robots have between two and six degrees of freedom. More degrees of freedom can be obtained by attaching independent systems together. An example is a multi-joint end-effector attached to a manipulator.

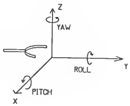

Figure 2-1. Six degrees of freedom of an end-effector

At any instance the placement of a robot with k degrees of freedom can be represented by a k-tuple. Figure 2-1 depicts the six degrees of freedom of an end-effector. The end-effector in this case can *translate* in the 3D space where its instantaneous positions are represented by its cartesian coordinates, *x*,*y* and *z*. It can also *rotate*: Its *orientation* at any point in time is represented by *roll*, *pitch*, and *yaw*, the rotations about the *y*,*x*, and *z* axes, respectively.

Robot work envelopes, the space bounded by the maximum reach of the manipulators, have different shapes. Based on the way the links are joined together, robots are grouped in different categories.

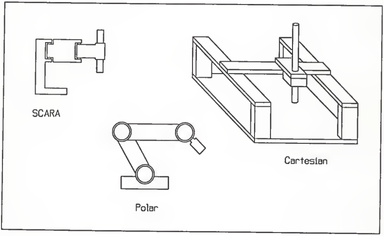

Figure 2-2. Common manipulator categories

Figure 2-2 represents three common manipulator categories. Cartesian robots have linear joints aligned along the cartesian axes. Their work envelope is a rectangular box. Polar robots usually have their joints represented by the polar coordinate system (r and theta). Their work envelope is hemi-spherical. The SCARA[1] category represents a combination of polar joints on horizontal planes and linear joints vertically oriented. The SCARA work envelope is cylindrical. These terms are commonly used but the boundaries between these categories are not clear since they are often combined. Although certain categories are better suited for certain purposes, for example polar robots are better for spray welding, cartesian and cylindrical robots for assembly, they can often be used interchangeably. In fact most systems can represent placements in the Cartesian coordinate system even though they are not of the Cartesian category.

The robot work space is usually three dimensional. For simplicity in certain problems the scope may be limited to a two dimensional view. An object moving in a 2-D plane may still have three degrees of freedom: Translation in the x and y directions on the horizontal plane and rotation about its vertical axis. In the rest of this paper we further limit the motions of the robot to two degrees of freedom by representing it by a point moving on a planar surface. Translation of a point object in the $x-y$ plane represents two degrees of freedom. Its rotations and orientation will be meaningless.

2.2 Representation of the robot work space

The robot environment is represented by a model of two-dimensional space containing a finite set of disjoint polygons points and connected line segments. The space boundary (the horizontal projection of the work envelope) and obstacles are represented by polygons. Obstacle polygons are disjoint and completely enclosed in the envelope polygon. Obstacles too close together may have to be merged and represented by one polygon. An obstacle located at the boundary may be "merged out" to the envelope polygon. The area outside the envelope and inside the obstacles is the forbidden region. The rest is the free space of the robot (also called *configuration space*).

1. SCARA stands for Selective Compliance Assembly Robot Arm.

2.3 Expanded-Obstacles approach

Motion of a single object in the presence of obstacles can be considered by shrinking the object to a point and enlarging the obstacles. We will use this method by Lozano-Perez and Wesley[9] to use a point to represent the robot end-effector which in real life can be of any shape.

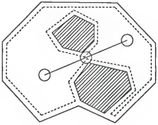

Figure 2-3. Example of Expanded Obstacles Representation

As a result, the obstacles are represented by enlarged polygons and similarly, the envelope polygon is shrunk down (Figure 2-3).

Positions of the robot (which is really the *end-effector* in this case, ignoring the rest of the manipulator)[2] are represented by its cartesian coordinates *(x,y)*. Connected line segments represent the robot paths. Obviously these lines are not allowed to cross the polygons, or collisions will occur.

2.4 Data structure representation

The objects, (points, polygons and segments) can be expressed as structures in the C programming

2. From this point we will use the terms *robot* and *end-effector* interchangeably to denote the position of the robot.

language as follows:

2.4.1 Points:

```
struct coord {
      float x;
      float y;
}
```

In real life, most robot systems maintain their own data structures representing points in space. They appear under the form of k-tuples for the k degrees of freedom of the manipulator as mentioned earlier. The two-member data structure of the points given here is necessary for the purpose of this paper but may be useless in real application.

2.4.2 Polygons:

```
struct polygon {
      int v_no;
      int closed;
      struct coord v[MAX_V];
}
```

In this structure v_*no* is the number of vertices of the polygon, v[ ] is the array of vertices ($v_i = (x_i, y_i)$). *Closed* is the status of the polygon. It can have a value of zero or equal to *v_no*. *V_no* starts with a value of zero and increments by one each time a vertex is entered when the polygon is being constructed. When the polygon is completed (*closed*) the last vertex in the array has the same coordinate values as the first, at which point *closed* is assigned the value of v_*no*. Thus, a (complete) polygon P of n vertices is an array of $n+1$ elements, $P = (v_0, v_1, ..., v_n)$ where

$$\begin{cases} x_{v_0} = x_{v_n} \\ y_{v_0} = y_{v_n} \end{cases}$$

and

$$closed = \text{v\_}no = n+1.$$

2.4.3 Segments:

```
struct segment {
      struct coord e1, e2;
      float a;
      float b;
}
```

Segments are not absolutely necessary to represent paths since they can simply be arrays of points. However this structure is included in the model for convenience in our following geometric computation. With the equation of a line, $y = ax + b$, where a is the *slope* and b, the *y–intercept*, we represent a line segment as a line bounded by two end points e_1 and e_2.

2.5 Some basic analytic geometry relations

At this point we take one step further to define a few formulae required for the path planning algorithms.

2.5.1 Equation of a line through two points

We need to determine a and b in the equation $y = ax + b$. With two points A and B we have the equation

$$\frac{y - y_A}{x - x_A} = \frac{y_B - y_A}{x_B - x_A}$$

from which we can deduce

$$a = \frac{\Delta y}{\Delta x}$$

and

$$b = y_A - ax_A$$

where $\Delta y = y_B - y_A$ and $\Delta x = x_B - x_A$. An exception is when $\Delta x = 0$, in which case the equation is represented by $x = y_A$

2.5.2 Intersection point of two lines

The intersection $I = (x_I, y_I)$ of two crossing lines y_1 and y_2 is the solution of the simultaneous equations

$$\begin{cases} y_1 = a_1 x + b_1 \\ y_2 = a_2 x + b_2 \end{cases}$$

The components x_I and y_I are derived as

$$x_I = \frac{b_2 - b_1}{a_1 - a_2}$$

and

$$y_I = a_2 x_I + b_2$$

except in the case of $a_1 = a_2$ where the lines are parallel and there is no intersection. (If $b_1 = b_2$ as well, the lines are super-imposed. This case will be treated as no intersection in this model.)

2.5.3 Length of a segment

The length of a segment *AB* which is the distance between point *A* and *B* is given by

$$|AB| = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

2.6 Limitations of the model

This model is only an approximation of two-dimensional space and confines the algorithms to the limits of a system with two degrees of freedom.

Representing natural objects with polygons usually requires approximation. The smaller the number of vertices (the less memory space) the less accurate the approximation. For obstacles the polygons are approximately equal to or larger than the real objects. For the outer boundary the approximation polygon has to fit inside the work envelope. As a result the free configuration space is reduced. If the work space is crowded with obstacles, the approximation needs to be very accurate. In the extreme case the model becomes useless because the representation would take too much memory space.

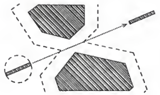

Figure 2-4. Expanded Obstacles representation of a rod

By representing the moving object with a point we lose control of its orientation. In using the Expanded Obstacle method the loss of free space is minimal if the object is a disc. For long and thin objects such as a rod, the waste of space is large (Figure 2-4).

Again, if the work space is too crowded, this loss of free space may be prohibitive. The solution in this case is to add another dimension to the representation of the moving object: Its orientation.

All models have their shortcomings. They are valuable only in their own context. Our model is designed to work in most practical cases where the robot has a reasonably large free configuration space.

3. DEVELOPMENT OF A PRACTICAL PATH-FINDING ALGORITHM

3.1 The REACH & CLEAR Algorithm

This is a fairly simple algorithm that will give a complete solution to the path finding problem. A thorough analysis will show that this solution is not the optimal solution in all cases but it is guaranteed to finding a complete path from any two points in the configuration space if such a path exists.

This algorithm involves a sequence of repeated calls to the two functions *Reach* and *Clear* which will give all the intermediate nodes to construct the complete path. Given a starting node, *Reach* determines whether the direct path from there to the destination point is clear. If it is, the destination point is reached. If is is not, *Reach* returns the coordinates of the first point where the path is blocked and the identification numbers of the blocking polygon and the correspondent segment. From that point *Clear* returns the subsequent vertices of the polygon ending with the vertex from which the current polygon is no longer an obstacle. Then *Reach* continues to find the next blockage and so on until the destination is reached and the path is complete.

3.1.1 Depth-first and Breadth-first searches

Obviously, *Clear* can return two possible solutions: A path continuing to the "left" and the other to the "right" of the polygons. Once given a point on a polygon, *Clear* uses the function *Next* to find the next vertex on the polygon. At some instances, *Next* returns the next higher index in the array of vertices of the polygon. At others, it returns the current or the next lower index in the array of vertices of the polygon. A parameter *dir* is set to "upper" or "lower" before each time *Clear* is executed. For a depth-first search *dir* is given a fixed value to guide *Next* in selecting the "upper" or the "lower" option throughout the entire process to find one path. (For one value of *dir*, a path may turn "left" at one obstacle and "right" at the next obstacle if the vertices entered in opposite directions, clockwise and counter-clockwise, when the corresponding polygons were being built. Paths constructed in both depth-first directions will be compared at the end, and the shortest one will be chosen.

This depth-first search method is successful in all cases consisting of convex polygons exclusively. For a work-space containing non-convex polygons solutions are not always guaranteed: If a polygon partially surrounds another, it may create region where the search path will become circular (and endless). Thus, breadth-first searches are required when non-convex polygons are involved. Breadth-first paths are obtained by constructing a binary tree in which branches consist of nodes found in both directions at each obstacle. A solution is guaranteed if the breadth-first method is used. However, it requires a lot more memory space than the depth-first method. One alternative approach is to represent non-convex polygons by smaller adjacent convex polygons and apply depth-first searches.

3.1.2 A Depth-first search function

Let us consider a depth-first function, *Findpath*, that constructs a complete path by alternatively calling *Reach* and *Clear*. Given the start and destination points L_0 and L_1, respectively, a path $P = N_n$ is to be constructed. N_n denotes the global array of nodes N_i in which the first element, $N_0 = L_0$ and the last element $N_n = L_1$. A global boolean variable, *pathcomplete*, is set to FALSE at the beginning of the process. A local boolean variable, *pathclear*, is used in *Reach*. Before the first call to *Reach*, n is assigned a value of zero. Each time a new node is determined, n is incremented by one. The variable *pathcomplete* is returned as TRUE and $N_n = L_1$ when L_1 is reached.

Findpath can be expressed in pseudo-codes as follows:

Findpath (dir):

Set *obstacle* $\leftarrow -1$

Set *pathcomplete* $\leftarrow$ *FALSE*

Set *pathclear* $\leftarrow$ *TRUE*

Repeat

{

pathcomplete $\leftarrow$ *Reach*(*obstacle*,*currentnode*:*obstacle*, *edge*, *nodeindex*)

If *pathcomplete* = *FALSE* then

currentnode $\leftarrow$ *Clear*(*obstacle*, *edge*, *nodeindex*)

} Until *pathcomplete*

The functions *Reach*, *Next* and *Clear* are described in pseudo-codes below:

Reach (*obstacle,currentnode*: *obstacle, edge, nodeindex*):

Let n be the next node index, $n \leftarrow nodeindex+1$

Let T_0 be the current node

Set $Count \leftarrow 0$

For all obstacle P_i such that i<>*obstacle* {

For all vertices V_j of polygon P_i {

Find all $i,\ j,\ S_{ij} = (T_0L_1 \cap V_jV_{j+1})$

where S_{ij} is the intersection of segments T_0L_1 and V_jV_{j+1},

i is the designation number of the obstacle

j is the designation number of the corresponding edge

If S_{ij} exists increment *Count*

}

}

If $Count \geq 2$ then {

Find $i,\ j,\ R_{ij}$ where

R_{ij} is the intersection closest to the current node

(T_0R_{ij} is the shortest of all segments T_0S_{ij}.)

Set $N_n \leftarrow R_{ij}$

Return: Obstacle P_i, edge j, nodeindex n

}

Else {

Set *pathcomplete* $\leftarrow$ *TRUE*

Set $N_n \leftarrow L_1$

}

Clear (*obstacle*, *edge*, *nodeindex*):

Let k be the next node index for the obstacle, $k \leftarrow nodeindex+1$

Let l be the vertex index,

$l \leftarrow edge$ for *lower* direction,

$l \leftarrow Next(edge)$ for *upper* direction

Set $pathclear \leftarrow FALSE$

Set N_k to the next vertex on the obstacle, $N_k \leftarrow V_l$

Repeat {

If the number of intersections of segment $N_k L_1$ with all segments $V_j V_{j+1}$,

$$\sum_{j=0}^{j=v\_no-1} (N_k L_1 \cap V_j V_{j+1}),$$

is greater than 2 then {

Set $k \leftarrow k+1$

Set $l \leftarrow Next(l)$

Set $N_k \leftarrow V_l$

}

Else {

Set $pathclear \leftarrow TRUE$

Set $nodeindex \leftarrow k$

}

} Until *pathclear*.

Return: All nodes N_k, *nodeindex* k

Next (*vertex*):

If $dir = upper$ then {

$$next \leftarrow \begin{cases} vertex < v\_no - 1: vertex + 1 \\ vertex \geq v\_no: 0 \end{cases}$$

}

Else {

$$next \leftarrow \begin{cases} vertex > 0: vertex - 1 \\ vertex = 0: v\_no - 1 \end{cases}$$

Return: *next*

3.1.3 Time complexity

Suppose, for the worst case of *Findpath* execution, n is the number of polygons, m is the largest number of vertices of any polygon, the time complexity of the above functions is estimated as follows:

Reach: $O(m \times n)$

Clear: $O(m)$

Findpath: $O(m \times n^3)$

3.2 Possible optimizations

An observation to be made about the *Reach* and *Clear* algorithm is that along the paths constructed there are situations where short-cuts are possible.

In situations where an obstacle is first "Reached", a node is set at the reach point. Then a subsequent node is set at the next corner of the obstacle. This corner node may be reached directly from the launching point if there is no obstacle in the way. If this short-cut is possible, the path will have less nodes and the total path length will be shorter. Even if there are obstacles between the launching node and the corner node, other intermediate nodes could be generated to obtain a shorter path. This kind of improvement may be built into *Reach* or may be done after a complete path is constructed.

Similar situations exist with *Clear* when the path surrounds non-convex obstacles. After an obstacle is reached, *Clear* generates nodes around the polygon until the path is cleared. If this occurs at a concave portion of the obstacle, extraneous nodes may be generated. Short-cuts should be sought between these nodes to optimize the path. Again, this optimization may be incorporated directly in *Clear* or may be part of a separate function executed after complete paths are generated.

Another kind of improvement could be made in *Reach.* Every time *Reach* is executed, it checks for possible intersections of the line segment from the current point to the destination point (T_0L_1) with all polygon segments. Since the polygons are stored in system memory as arrays, there is no indication that an obstacle may be "behind" the current point. Thus, the number of check points is not reduced after an obstacle has been visited. A solution is to "mark" the polygons when they are being checked so that they will not be checked again in the same process. Although this may improve the response time, it will require more memory. The gain in the response time may not be significant enough to justify the additional memory use.

3.3 Direct applications

This algorithm is based on the proposed two-dimensional model and is primarily a theoretical solution. However, despite its simplicity it may be applied to certain real life applications without (or with little) modifications.

The closest applications would be in manufacturing assembly processes using certain types of SCARA and Cartesian robots. As described earlier, some of these robots have a vertically oriented linear axis (cylindrical and rectangular work envelope). The cases of interest are when the robot end-effector is allowed to move on a horizontal surface below the height where the intermediate links of the arm's

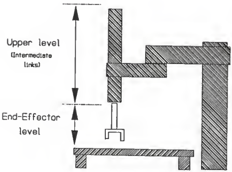

Figure 3-1. A SCARA robot application

components are located (Figure 3-3). Assuming that these intermediate links can move around the upper horizontal plane without obstruction, the multiple-link clearance question may be ignored. This reduces the complex motion planning problem to that of a single moving object. Moreover, this allows us to ignore the height component of the three-dimensional space in most cases. The scope of the motion problem can be reduced to two-dimensional as the model represents.

The next closest application foreseeable is for AGV's (Automatic Guided Vehicles). This type of application of the model seems to be even more feasible since these vehicles travel on a two-dimensional horizontal plane (i.e., the ground). The problem is with today's technology, most of these AGV's are used with fixed guiding path on the factory floor[10] The AGV's are often allowed to travel (at limited speed) in the same area where human workers are since their paths are fixed. Applying the *Reach* and *Clear* algorithm for AGV's on the human populated factory floor may cause safety problems since moving obstacles (human operators) are not known by the AGV's and their paths would be unpredictable.

4. SIMULATION PROGRAM

The simulation program is based on the two-dimensional model described, is implemented in the C programming language, and runs on the MS-DOS operating system. The program creates an interactive environment to allow easy creation of different configurations of obstacles in which path finding algorithms are tested. The user/developer selects options from the command menu via the keyboard and draws obstacles on the video monitor screen with a mouse. The *Reach* and *Clear* algorithm is built in the simulation and is ready for testing. The program is organized so that other algorithms can be developed and tested in the same environment. Although this requires part of the program to be modified and the program to be recompiled, the program modules are organized so that new functions can be added to the menu conveniently. A program listing is included in the appendix.

4.1 Organization

The program is organized into a menu tree with a user interface consisting of keyboard and mouse input and graphics display. A high resolution graphics adapter (EGA or VGA)[3] and the Microsoft Mouse device driver are used. At the beginning of the execution, the main program verifies availability of a video graphics adapter and the mouse device driver and initializes them before setting up the main menu. The program is organized into a hierarchy of modules making up the branches in the menu tree. The modules are maintained separately and linked together by a MAKE script. Below is the list and description of the modules:

— Findp.c: This is the main module. It sets up the main menu and allows calling other modules.

— Obstacle.c: This module allows the drawing of polygons to represent the obstacles.

3. Enhanced Graphics Adapter and Video Graphics Array, respectively

— Linesegm.c: This is the "tool box" containing various functions used by the algorithms.

— Setpoint.c: This module allows the user to set the start and destination points for testing.

— Storage.c: This module takes care of the loading and saving of obstacles configurations from and to data files.

— Walk.c: This is the collection of "algorithms". It allows testing of these algorithms on different configurations.

The menu tree (Figure 4-1) consists of commands to describe the configurations (Obstacle, Setpoint), to load and save different configurations (File) and to test the algorithms (Run, Walk). Command selections are made by entering the capital letter of the command word (for example "B" in oBstacle). Menu selections are entered via the keyboard only. Some commands in the main session (in which the main menu is active) may invoke lower level sessions where corresponding menus will be displayed. These menus provide an option to go back to the previous level when the session is finished. Program execution stops when the "Quit" option of the main menu is selected and confirmed.

The display screen is a two-dimensional matrix of 640x350 pixels (640x480, for VGA mode). The menu occupies the top 20 pixel-lines. The rest of the screen represents a rectangular robot work envelope. Obstacles, locations and paths are displayed in different colors. The mouse is used to draw obstacles and to position the start and destination points. The mouse cursor movements are limited within the display of the work envelope. When the appropriate session is active, points can be entered with the left mouse button. The right mouse button is used to refresh the screen at most levels.

4.2 Functions

Below is a list of functions in the menu tree along with their brief description.

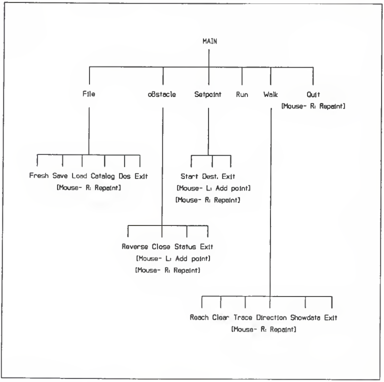

Figure 4-1. Menu tree of the simulation program

- FILE:

— Fresh: Clear work-space in memory of all objects to re-start.

— Save: Save current configuration (all existing obstacles) in a data file.

— Load: Load a saved configuration from a data file. The current configuration will be over-written.

— Catalog: Show a list of all saved configuration data files.

— Dos: Execute a system level command.

— Exit: Go back to the main menu.

- OBSTACLE: Vertices are entered by clicking the Left mouse button. The Right button is to repaint the screen.

  — Reverse: Remove the last vertex entered (and the corresponding edge).

  — Close: Close the loop and complete the obstacle.

  — Status: List all the vertices entered for the current obstacle.

  — Exit: Go back to the main menu. A re-confirmation is required.

- SETPOINT: Select Start or Destination

  — Start: Enter the Start point by clicking the left mouse button. A small white circle indicates the resulting point.

  — Destination: Enter the Destination point by clicking the left mouse button. A small yellow circle indicates the resulting point.

  — Exit: Go back to the main menu.

- RUN: Select and execute path-finding programs based on different algorithms.

- WALK: Step-by-step walk-through the path-finding process.
  - — Reach: Execute the Reach function from the current node.
  - — Clear: Execute the Clear function from the current node.
  - — Trace: Draw a path from the Start point to the current node.
  - — Direction: Select or de-select the *upper* direction.
  - — Showdata: Turn on/off the show-data mode. If it is on, progress data will be displayed. Exit: Go back to the main menu.

- Quit: Leave the interactive environment. All configuration data will be lost unless saved in a data file.

4.3 Usage

The program is menu driven and easy to use. The user simply selects options on the menu with single keystrokes and follows the brief instructions on the menu line.

To enter the program, the executable program name "findp" must be entered at the operating system level. An EGA or VGA graphics adapter and a mouse are assumed to be available. The mouse device driver must be installed before *findp* can be executed or an error message ("Mouse not installed") will appear.

To exit the program normally, "Quit" option on the main menu must be selected and confirmed. The program can also be interrupted anytime with the <Control-C> keystroke combination. However, this is not recommended since the display screen may be left at an unwanted video mode after the program is interrupted.

4.4 Portability note

A special objective of the simulation is to keep the algorithm as system-independent as possible. Therefore in the simulation program design, the use of system specific library functions are limited to those absolutely necessary to simulate the environment and not to help solve problems in the path finding algorithm. Specifically, the most library functions used in the simulation are graphics display functions: Line drawing, color setting, etc. For instance, a possible means to determine if a line intersects with a polygon is by using color codes. First the area inside the polygon (all pixels within the polygon boundary) is given a specific color. This may be done using a "flood fill" graphics library function. The line is assigned a different color. From this point the intersection point may be determined by moving along the line until the polygon color is found. Color coding is not impossible in real applications. However, not all systems have this capability. Therefore this coding scheme is avoided in the simulation program in order to maintain the fidelity with the real applications.

5. EXTENSIONS, FURTHER STUDIES

Extensions of this project could include more use of the advances mentioned in the survey if the overhead/performance trade-off remains practical.

Representation of non-zero radius and oriented moving objects is the most related problem outside the scope of this project. It would be a direct extension of the 2D model to solve the limitation problem described in Chapter 2. Essentially, a third degree of freedom of the moving object (the rod in the Figure 2-4) is required to represent its orientation in addition to its position: (x, y, θ).

Other foreseeable extensions are numerous and may require substantial modifications to the model : Multilink manipulators, moving obstacles, three-dimensional environment, etc.

REFERENCES

1. *VAL-II Reference guide*, Version 6.0, Adept Technology, Inc., March 1987
2. Pinson, E., "Modeling and Simulation for Robot Software Development" *Japan U.S.A Symposium on Flexible Automation*, July 1986, pp. 615-619.
3. Sharir, M., "Algorithmic Motion Planning in Robotics", *I.E.E.E. Computer*, Vol. 22, No. 3, March 1989, pp. 9-20.
4. Schwartz, J. T. and Sharir, M., "On the Piano Movers' Problem: II," *Advances in Applied Math.*, Vol. 4, 1983, pp. 298-351.
5. Canny, J., *Complexity of Robot Motion Planning*, Doctoral dissertation, MIT, Cambridge, Mass., 1987.
6. Leven, D., and Sharir, M., "An Efficient and Simple Motion-Planning Algorithm for a Ladder Moving in Two-Dimensional Space Amidst Polygonal Barriers," *J. Algorithms*, Vol. 8, 1987, pp. 192-215.
7. Hopcroft, J. E. and Wilfong, G. T., "Reducing Multiple Object Motion Planning to Graph Searching," *SIAM J. COMPUT.*, Vol. 15, No. 3, August 1986, pp. 768-785.
8. O'Dunlaing, C., and Yap, C., "A Retraction Method for Planning the Motion of a Disc," *J. Algorithms*, Vol. 6, 1985, pp. 104-111.
9. Lozano-Perez, T. & Wesley, M., "An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles" *Comm. ACM*, Vol. 22, 1979, pp. 560-570.
10. Nelson, W. L. and Cox, I. J., "Local Path Control for an Autonomous Vehicle", *Proceedings, I.E.E.E. International Conference on Robotics and Automation*, April 1988, pp. 1504-1510.

APPENDIX: Program Listing

```
/* Makefile: findp */
Findp.obj: Findp.c
	qcl /c /AM Findp.c

Obstacle.obj: Obstacle.c
	qcl /c /AM Obstacle.c

Setpoint.obj: Setpoint.c
	qcl /c /AM Setpoint.c

Walk.obj: Walk.c
	qcl /c /AM Walk.c

Linesegm.obj: Linesegm.c
	qcl /c /AM Linesegm.c

Storage.obj: Storage.c
	qcl /c /AM Storage.c

findp.exe: Findp.obj Obstacle.obj Setpoint.obj Walk.obj Linesegm.obj Storage.obj
	link Findp.obj+Obstacle.obj+Setpoint.obj+Walk.obj+Linesegm.obj	+Storage.obj;

/* findp.h */
#include <dos.h>
#include <stdio.h>
#include <graph.h>
#include <math.h>
#include <conio.h>

#define INFIN	0
#define FALSE	0
#define TRUE	1

#define MOUSE_IO	51
#define INIT_MOUSE	0
#define SHOW_CURSOR 1
#define HIDE_CURSOR	2
#define READ_MOUSE	3
#define SET_POS			4
#define X_LIMITS	7
#define Y_LIMITS	8

#define MAX_OBST	20
#define MAX_VRTX	100
```

```
#define BLK     0
#define BLU     1
#define GRN     2
#define CYA     3
#define RED     4
#define MAG     5
#define BRN     6
#define WHT     7
#define GRY     8
#define LTBLU   9
#define LTGRN   10
#define LTCYA   11
#define LTRED   12
#define LTMAG   13
#define YEL     14
#define LTWHT   15

union REGS inregs, outregs;
struct videoconfig vc;
struct coord {
        float x;
        float y;
};
struct polygon {
        int v_no;         /* no. of vertices */
        int closed;       /* =v_no if closed, =0 if not */
        struct coord v[MAX_VRTX];
};
struct segment {
        struct coord e1;
        struct coord e2;
        float a; /* slope */
        float b; /* y intercept */
};


/* Module: findp.c */

#include "findp.h"
char *cmd_msg;
char main_mnu[]= ("COMMAND: File oBstacle Setpoint Run Walk Quit
 [Right button: Repaint]");
char file_mnu[]= ("FILE: Fresh Save Load Catalog Dos Exit");
char setpoint_mnu[]= ("SETPOINT: Start Destination Exit");
char point_mnu[]= ("[Left button: Set Start/Destination point]");
char obstacle_mnu[]= ("OBSTACLE: Reverse Close Status Exit
 [BUTTONS - L: Enter vertices - R: Repaint]");
char walk_mnu[]= ("WALK: Reach Clear Trace Direction Showdata Exit
```

```
 [Right button: Repaint]");

int w_limit, e_limit, n_limit, s_limit;
int c= ' ', num= 0;

struct polygon obj[MAX_OBST];
struct coord loc[3], tmp[3], node[200];

/**********************************************************************/
main() {
/**********************************************************************/
        loc[0].x= 0; loc[1].x= 0;
        inregs.x.ax= INIT_MOUSE;
        int86(MOUSE_IO, &inregs, &outregs);
        if (outregs.x.ax == 0){
                printf("Mouse not installed0 );
                exit(0);
        }
        if (_setvideomode(_VRES16COLOR) == 0) {
                if(_setvideomode(_ERESCOLOR) == 0) {
                        printf("No EGA/VGA available0);
                        exit(0);
                }else printf("EGA mode0);
        } else printf("VGA mode0);
        /* Just flash this on the screen */
        _getvideoconfig(&vc);
        w_limit= 0;
        n_limit= 20;
        e_limit= vc.numxpixels-1;
        s_limit= vc.numypixels-1;
        _setcolor(GRN);
        _rectangle(_GBORDER, w_limit, n_limit, e_limit, s_limit);

        inregs.x.cx= w_limit+2; inregs.x.dx= e_limit-3;
        inregs.x.ax= X_LIMITS;
        int86(MOUSE_IO, &inregs, &outregs);

        inregs.x.cx= n_limit+2; inregs.x.dx= s_limit-2;
        inregs.x.ax= Y_LIMITS;
        int86(MOUSE_IO, &inregs, &outregs);

        inregs.x.ax= SHOW_CURSOR;
        int86(MOUSE_IO, &inregs, &outregs);
        cmd_msg= main_mnu;
        Repaint();

        for (;;) {
                Buttons();
```

```
		if (kbhit()){
			c= tolower(getch());
			if (c == 'f') File();
			if (c == 'b') Obstacle(&obj[num]);
			if (c == 's') Setpoint();
			if (c == 'r') {    /* Run */
				if ((loc[0].x == 0) || (loc[1].x == 0))
					printf("Start/Destination points unknown0");
				else
					if (num == 0)
						printf("No obstacles entered0");
					else
						Run();
			}
			if (c == 'w') {    /* Walk */
				if ((loc[0].x == 0) || (loc[1].x == 0))
					printf("Start/Destination points unknown0");
				else
					if (num == 0)
						printf("No obstacles entered0");
					else
						Walk();
			}
			if (c == 'q') {
				printf("Are You Sure? [n]");
				c= getch();
				if (c == 'y') break;
				Repaint();
			}
		}
	}
	_clearscreen(_GCLEARSCREEN);
	_setvideomode(_DEFAULTMODE);
} /* main */

/***********************************************************************/
Buttons() {
/***********************************************************************/
		inregs.x.ax= READ_MOUSE;
		int86(MOUSE_IO, &inregs, &outregs);
		if (outregs.x.bx & 0x2) {  /* Right button */
			while (outregs.x.bx & 0x2) {
				inregs.x.ax= READ_MOUSE;
				int86(MOUSE_IO, &inregs, &outregs);
			}
			Repaint();
		}
		if (outregs.x.bx & 0x1) {  /* Left one not used */
```

```
                        printf("Keyboard menu selection only0");
                        while (outregs.x.bx & 0x1) {
                                inregs.x.ax= READ_MOUSE;
                                int86(MOUSE_IO, &inregs, &outregs);
                        }
                        Repaint();
                }
} /* Buttons */


/* Module:  Linesegm.c */

#include "findp.h"
extern int num;
extern struct coord tmp[];
extern struct polygon obj[];
/*********************************************************************/
Crosscount() {
/*********************************************************************/
        int i, j;
        int hitcount;
        struct coord h;

        hitcount=0;
        for (i= 0; i < num; i++)
                for (j= 0; j < obj[i].v_no; j++) {
                        if (Cross(&h,&tmp[0],&tmp[1],&obj[i].v[j],&obj[i].v[j+1])) {
                                hitcount++;
        /*DIAGNOSTICS*/
                                _moveto(h.x, h.y);
                                _setcolor(LTMAG);
                                _setpixel(h.x, h.y);
                                _ellipse( _GBORDER, h.x -3, h.y -3, h.x +3, h.y +3);
        /*DIAGNOSTICS*/
                        }
                }
        return(hitcount);
} /* Crosscount */

/*********************************************************************/
int Cross(junction, p1, p2, w1, w2)
/*********************************************************************/
struct coord *junction, *p1, *p2, *w1, *w2;
{
struct segment pline, wline;
float xi, yi;
int xonw, xonp, yonw, yonp;
        Line_eq(&pline, p1->x, p1->y, p2->x, p2->y);
```

```
    Line_eq(&wline, w1->x, w1->y, w2->x, w2->y);

    if (pline.a == wline.a) return(FALSE); /* Parallel */

    if ((pline.a == INFIN) && (p1->x == p2->x)) {       /* Vertical */
            xi= (p1->x);
            yi= (wline.a * xi + wline.b);
    } else
    if ((wline.a == INFIN) && (w1->x == w2->x)) {     /* Vertical */
            xi= (w1->x);
            yi= (pline.a * xi + pline.b);
    } else {
            xi= ((wline.b - pline.b) / (pline.a - wline.a));
            yi= (wline.a * xi + wline.b);
    }

    if (w1->x < w2->x) {
            xonw= ( ((w1->x -.3 <= xi) && (xi <= w2->x +.3)) ? 1 : 0);
    } else {
            xonw= ( ((w2->x -.3 <= xi) && (xi <= w1->x +.3)) ? 1 : 0);
    }

    if (p1->x < p2->x) {
            xonp= ( ((p1->x -.3 <= xi) && (xi <= p2->x +.3)) ? 1 : 0);
    } else {
            xonp= ( ((p2->x -.3 <= xi) && (xi <= p1->x +.3)) ? 1 : 0);
    }


    if (w1->y < w2->y) {
            yonw= ( ((w1->y -.3 <= yi) && (yi <= w2->y +.3)) ? 1 : 0);
    } else {
            yonw= ( ((w2->y -.3 <= yi) && (yi <= w1->y +.3)) ? 1 : 0);
    }

    if (p1->y < p2->y) {
            yonp= ( ((p1->y -.3 <= yi) && (yi <= p2->y +.3)) ? 1 : 0);
    } else {
            yonp= ( ((p2->y -.3 <= yi) && (yi <= p1->y +.3)) ? 1 : 0);
    }

    if (xonw && xonp && yonw && yonp) {
            junction->x= xi;
            junction->y= yi;
            return (TRUE);
    } else return (FALSE);
} /* Cross */
```

```
Round (fval)
float fval;
{
        return ( ((fmod(fval, 1.0)) >= .5) ? ceil(fval) : floor(fval) );
}

/*******************************************************************/
Line_eq(line, x1,y1, x2,y2)
/*******************************************************************/
struct segment *line;
float x1, y1, x2, y2;
{
        float deltax, deltay;
        /*DIAGNOSTICS
        printf("Line_eq: x1=%f y1=%f, x2=%f y2=%f0, x1,y1, x2,y2);
        DIAGNOSTICS*/
        deltax= x2 - x1;
        if (deltax == 0) {
                line->a= INFIN; /* Infinity : Vertical*/
        } else {
                deltay= y2 - y1;
                line->a= deltay/deltax;
                line->b= y1 - (deltay/deltax) * x1;
        }
} /* Line_eq */


/* Module:  Obstacle.c */

#include "findp.h"
extern int num;
extern char *cmd_msg;
extern char obstacle_mnu[], main_mnu[];
extern struct coord loc[], tmp[];
/*******************************************************************/
Obstacle(W)
/*******************************************************************/
struct  polygon *W;
{
int c= ' ', j;
int count;
        W->v_no= 0; W->closed= 0;
        cmd_msg= obstacle_mnu;
        Repaint();
        _setcolor(LTRED);
        for (;;){
                inregs.x.ax= READ_MOUSE;
                int86(MOUSE_IO, &inregs, &outregs);
```

```
if (outregs.x.bx & 0x1){
        tmp[0].x= outregs.x.cx;
        tmp[0].y= outregs.x.dx;
        tmp[1].x= 0;
        tmp[1].y= 0;
        if (W->v_no == 0){        /* New polygon */
                if ((Crosscount() % 2) == 0) {
                        _moveto(tmp[0].x, tmp[0].y);
                        _setpixel(tmp[0].x, tmp[0].y);
                        W->v[W->v_no].x= tmp[0].x;
                        W->v[W->v_no].y= tmp[0].y;
                        W->v_no++;
                } else {
                        printf("Illegal point inside obstacle0);
                }
        } else {            /* Same polygon */
                tmp[1].x= W->v[W->v_no-1].x;
                tmp[1].y= W->v[W->v_no-1].y;
                if ((tmp[0].x != tmp[1].x) || (tmp[0].y != tmp[1].y)) {
                        if (Crosscount() == 0) {
                                _lineto(tmp[0].x, tmp[0].y);
                                W->v[W->v_no].x= tmp[0].x;
                                W->v[W->v_no].y= tmp[0].y;
                                W->v_no++;
                        } else {
                                printf("Non disjoint obstacles0);
                        }
                }
        }
        while (outregs.x.bx & 0x1) {
                inregs.x.ax= READ_MOUSE;
                int86(MOUSE_IO, &inregs, &outregs);
        } /* Button released */
}
if (outregs.x.bx & 0x2) {  /* Repaint */
        while (outregs.x.bx & 0x2) {
                inregs.x.ax= READ_MOUSE;
                int86(MOUSE_IO, &inregs, &outregs);
        }
        Repaint();
}
if (kbhit()){
        c= tolower(getch());
        if (c == 'e') {     /* Exit -- Abort */
                printf("Abort? [n]");
                c= getch();
                if (c == 'y') {
                        W->v_no= 0;
```

```
                        cmd_msg= main_mnu;
                        Repaint();
                        break;
                }
                Repaint();
        }
        if ((c == 'r') && (W->v_no > 0)) {              /* Reverse */
                _setcolor(BLK);
                (W->closed > 0) ? W->closed= 0 : W->v_no--;
                _lineto(W->v[W->v_no-1].x, W->v[W->v_no-1].y);
                _setcolor(LTRED);
                if (W->v_no == 1) _setpixel(W->v[0].x, W->v[0].y);
        }
        if ((c == 'c') && (W->v_no > 2)) {
                tmp[0].x= W->v[0].x;
                tmp[0].y= W->v[0].y;
                tmp[1].x= W->v[W->v_no-1].x;
                tmp[1].y= W->v[W->v_no-1].y;
                if (Crosscount() == 0) {
                        W->v[W->v_no].x= W->v[0].x;
                        W->v[W->v_no].y= W->v[0].y;
                        _lineto(W->v[0].x, W->v[0].y);
                        W->closed= W->v_no;
                        num++;

                        tmp[0].x= loc[0].x;
                        tmp[0].y= loc[0].y;
                        tmp[1].x= loc[1].x;
                        tmp[1].y= loc[1].y;
                        if ((Crosscount() % 2) != 0) {
                                printf("No setpoints allowed in obstacle0);
                                printf("[Repaint and continue]0);
                                num--;
                                W->closed= 0;
                        } else {

                        cmd_msg= main_mnu;
                        Repaint();
                        break; /* Polygon completed */
                        }
                } else {
                        printf("No overlapped obstacles allowed0);
                }
        }
        if (c == 's') {    /* Status */
                printf("Object #%d: %3d points entered0,
                        num+1, W->v_no);
                for(j= 0; j < W->v_no; j++)
```

```
                                        printf("j=%2dx=%.2f y=%.2f0,
                                                j, W->v[j].x, W->v[j].y);
                        }
                }
        }
} /* Obstacle */


/* Module:  Setpoint.c */

#include "findp.h"
extern char *cmd_msg;
extern char setpoint_mnu[], point_mnu[], main_mnu[];
extern struct coord tmp[], loc[];

/**********************************************************************/
Setpoint() {
/**********************************************************************/
        int c;
        cmd_msg= setpoint_mnu;
        Repaint();
        for (;;) {
                Buttons();
                if (kbhit()) {
                        c= tolower(getch());
                        if (c == 's') {
                                Point(&loc[0]);
                                break; /* for */
                        }
                        if (c == 'd') {
                                Point(&loc[1]);
                                break; /* for */
                        }
                        if (c == 'e') break; /* for */

                }
        }
        cmd_msg= main_mnu;
        Repaint();
} /* Setpoint */

/**********************************************************************/
Point(spot)
/**********************************************************************/
struct coord *spot;
{
int count;
        cmd_msg= point_mnu;
```

```
        Repaint();
        for (;;){
                inregs.x.ax= READ_MOUSE;
                int86(MOUSE_IO, &inregs, &outregs);
                if (outregs.x.bx & 0x1){
                        while (outregs.x.bx & 0x1) {
                                inregs.x.ax= READ_MOUSE;
                                int86(MOUSE_IO, &inregs, &outregs);
                        }
                        tmp[0].x= outregs.x.cx;
                        tmp[0].y= outregs.x.dx;
                        tmp[1].x= 0;
                        tmp[1].y= 0;
                        count= Crosscount();
                        if ((count % 2) == 0) {
                                spot->x= tmp[0].x;
                                spot->y= tmp[0].y;
                                _setcolor(LTWHT);
                                _moveto(spot->x, spot->y);
                                _setpixel(spot->x, spot->y);
                        _ellipse( _GBORDER, spot->x -5, spot->y -5,
                                        spot->x +5, spot->y +5);
                        break;
                        } else {
                                printf("Illegal point inside obstacle0);
                        }
                }
        }
} /* Point */


/* Module:  Storage.c */

#include "findp.h"
extern int w_limit, n_limit, e_limit, s_limit, num;
extern char *cmd_msg;
extern char file_mnu[], main_mnu[];
extern struct coord loc[];
extern struct polygon obj[];
/*********************************************************************/
File () {
/*********************************************************************/
char cmd[100];
int c;
        cmd_msg= file_mnu;
        Repaint();
        for (;;)
                if (kbhit()) {
```

```
                        _setvideomode(_TEXTC80);
                        c= tolower(getch());
                        if (c == 'f') {     /* Fresh */
                                printf("Clear work-space? [n]");
                                c= tolower(getch());
                                if (c == 'y') {
                                        num= 0;
                                        loc[0].x= 0;
                                        loc[1].x= 0;
                                        obj[0].v_no= 0;
                                        obj[0].closed= 0;
                                }
                                break;
                        }
                        if (c == 'l') {                 /* Load */
                                Load();
                                break;
                        }
                        if (c == 's') {                 /* Save */
                                if (num > 0) {
                                        Save();
                                        break;
                                } else printf("No Obstacles to save0);
                        }
                        if (c == 'c') {     /* Catalog */
                                system("dir *.dat");
                                printf("Hit a key to resume");
                                c= getch();
                                break;
                        }
                        if (c == 'd') {                 /* Dos */
                                printf("DOS command: ");
                                gets(cmd);
                                system (cmd);
                                printf("Hit a key to resume");
                                c= getch();
                                break;
                        }
                        if (c == 'e') break; /* Exit */
                }
        if (_setvideomode(_VRES16COLOR) == 0)
                _setvideomode(_ERESCOLOR);
        cmd_msg= main_mnu;
        Repaint();
        inregs.x.ax= SHOW_CURSOR;
        int86(MOUSE_IO, &inregs, &outregs);
} /* File */
```

```
/***********************************************************************/
Load () {
/***********************************************************************/
FILE *stream;
char fname[20];
int i, j;
float number;
        system("dir *.dat");
        printf("Oata file to read (no extension) [! to abort]: ");
        gets(fname);
        if (strcmp(fname,"!") == 0) {
                Repaint();
                return;
        }
        strcat(fname,".dat");
        if ((stream= fopen(fname,"r")) == NULL)
                printf("Could not open %s for loading0,fname);
        else {
                num= 0;         /* Clear work-space */
                loc[0].x= 0;
                loc[1].x= 0;
                obj[0].v_no= 0;
                obj[0].closed= 0;

                fscanf(stream, "%d", &num);
                printf("Num= %d0,num);
                for (i= 0; i < num; i++) {
                        fscanf(stream, "%d", &obj[i].v_no);
                        printf("V_no= %d0,obj[i].v_no);
                        obj[i].closed= obj[i].v_no;
                        for (j=0; j <= obj[i].v_no; j++) {
                                fscanf(stream, "%f", &obj[i].v[j].x);
                                fscanf(stream, "%f", &obj[i].v[j].y);
                        }
                }
                obj[i].v_no=0;
                fcloseall();
        }
} /* Load */

/***********************************************************************/
Save() {
/***********************************************************************/
FILE *stream;
char fname[20];
int  i, j;
        system("dir *.dat");
        printf("10ave to (file name with no extension) [! to abort]: ");
```

```
        gets(fname);
        if (strcmp(fname,"!") == 0) {
                Repaint();
                return;
        }
        strcat(fname,".dat");
        if ((stream= fopen(fname,"w")) == NULL)
                printf("Could not open %s0,fname);
        else {
                fprintf(stream, "%d0, num);
                printf("Num= %d0,num);
                for (i= 0; i < num; i++) {
                        fprintf(stream, "%d0, obj[i].v_no);
                        for (j=0; j <= obj[i].v_no; j++) {
                                fprintf(stream, "%.1f", obj[i].v[j].x);
                                fprintf(stream, "%.1f0, obj[i].v[j].y);
                        }
                }
                fcloseall();
        }
        printf("Saved to %s, %d obstacles0, fname, num);
} /* Save */

/************************************************************************/
Repaint() {
/************************************************************************/
        int i, j;
        inregs.x.ax= HIDE_CURSOR;
        int86(MOUSE_IO, &inregs, &outregs);
        _clearscreen(_GCLEARSCREEN);
        _setcolor(GRN);
        _rectangle(_GBORDER, w_limit, n_limit, e_limit, s_limit);
        printf("%s0, cmd_msg);

        if (loc[0].x != 0) {
                _setcolor(LTWHT);
                _moveto(loc[0].x, loc[0].y);
                _setpixel(loc[0].x, loc[0].y);
                _ellipse( _GBORDER, loc[0].x -5, loc[0].y -5,
                                loc[0].x +5, loc[0].y +5);
        }
        if (loc[1].x != 0) {
                _setcolor(YEL);
                _moveto(loc[1].x, loc[1].y);
                _setpixel(loc[1].x, loc[1].y);
                _ellipse( _GBORDER, loc[1].x -5, loc[1].y -5,
                                loc[1].x +5, loc[1].y +5);
        }
```

```
        _setcolor(LTRED);
        for (i= 0; i <= num; i++){
                _moveto(obj[i].v[0].x, obj[i].v[0].y);
                for(j= 1; j < obj[i].v_no; j++)
                        _lineto(obj[i].v[j].x, obj[i].v[j].y);
                if (obj[i].closed > 0) _lineto(obj[i].v[0].x, obj[i].v[0].y);
                if (obj[i].v_no== 1) _setpixel(obj[i].v[0].x, obj[i].v[0].y);
        }

        inregs.x.ax= SHOW_CURSOR;
        int86(MOUSE_IO, &inregs, &outregs);
} /* Repaint */


/* Module: Walk.c */

#include "findp.h"
extern int num;
extern struct coord loc[], node[];
extern struct polygon obj[];

extern char *cmd_msg;
extern char walk_mnu[], main_mnu[];

struct crosspoint {
        int oid, lid;
        struct coord p;
        float dist;
};
struct crosspoint spot[50];

int n, obst, edge;
int pathclear, pathcomplete, show, upper; /* Booleans */

/*********************************************************************/
Run () {
/*********************************************************************/
        int c;
        cmd_msg= main_mnu;
        Repaint();

        node[0].x=loc[0].x;
        node[0].y=loc[0].y;
        n= 0; /* first node */
        obst= -1; /* init */
        pathcomplete= FALSE;
        pathclear= TRUE;
        show= FALSE;
```

```
        printf("10elect algorithm:0);
        printf("[1]Upper Depth-first Reach&Clear0);
        printf("[2]Lower Depth-first Reach&Clear0);
        printf("[ ]Breadth-first Reach&Clear0);
        printf("[ ]Optimizing Breadth-first Reach&Clear0);
        printf("[ ]Optimizing Breadth-first Reach&Clear0);
        c= (getch());
        if (c == '1') {
                Repaint();
                printf("Upper Reach&Clear0);
                upper= TRUE;
                do {
                        Reach();
                        Clear();
                } while (!pathcomplete);
        }
        if (c == '2') {
                Repaint();
                printf("Lower Reach&Clear0);
                upper= FALSE;
                do {
                        Reach();
                        Clear();
                } while (!pathcomplete);
        }
        if (c == ' ') Repaint();
} /* Run */

/**********************************************************************/
Walk () {
/**********************************************************************/
        int c;
        cmd_msg= walk_mnu;
        Repaint();

        pathcomplete= FALSE;
        pathclear= TRUE;
        node[0].x=loc[0].x;
        node[0].y=loc[0].y;
        n= 0;    /* first node */
        obst= -1; /* init */

        for (;;) {
                Buttons();
                if (kbhit()) {
                        c= tolower(getch());
                        if (c == 'd') {
                                printf("Select Upper direction? [n]");
```

```
                    if (tolower(getch()) == 'y') {
                            upper= TRUE;
                            prinıf("0pper direction selected0);
                    } else {
                            upper= FALSE;
                            prinıf("7920ower direction selected0);
                    }
            }
            if (c == 's') {
                    prinıf("Select Showdata mode? [n]");
                    if (tolower(getch()) == 'y') {
                            show= TRUE;
                            printf("48rocess data will be shown0);
                    } else {
                            show= FALSE;
                            prinıf("48rocess data will NOT be shown0);
                    }
            }
            if (c == 'r')
                    if (!pathcomplete)
                            Reach();
                    else
                            prinıf("PATH COMPLETE0);
            if (c == 'c')
                    if (!pathcomplete)
                            Clear();
                    else
                            printf("PATH COMPLETE0);
            if (c == 't') Trace();
            if (c == 'z') Zip();
            if (c == 'e') { /* Exit */
                    cmd_msg= main_mnu;
                    Repaint();
                    break;
            }
        }
    } /* for */
} /* Walk */

/**********************************************************************/
Reach () {
/**********************************************************************/
        struct coord tmp;
        int i, j;
        int count, hit;
        float temp;
        struct coord h;
```

```
if (pathcomplete) return;
if (!pathclear) {
        printf("Reach done.  Try Clear0);
        return;
}
tmp.x= node[n].x;
tmp.y= node[n].y;
if ((n == 0) ||                ((node[n].x != node[n-1].x)||(node[n].y != node[n-1].y))
        ) n++;
if (show) {
        printf("0each: Finding node#%d0,n);
        printf(upper?"Upper direction0:"Lower direction0);
        /*Repaint();*/
}

count= 0;
for (i= 0; i < num; i++)
        if (i != obst)
                for (j= 0; j < obj[i].v_no; j++)
                        if (Cross(&h, &tmp, &loc[1],
                                        &obj[i].v[j], &obj[i].v[j+1])) {
                                spot[count].oid= i;
                                spot[count].lid= j;
                                spot[count].p.x= h.x;
                                spot[count].p.y= h.y;
                                spot[count].dist= sqrt(     pow(h.x - tmp.x, 2) +
                                        pow(h.y - tmp.y, 2) );
                                count++;
                        }
if (show) printf("Cross: %d, ",count);
if (count >= 2) {
/*|| ((count == 2 )&&(spot[0].p.x != spot[1].p.x))) */
        temp= spot[0].dist;
        hit= 0;
        for (i= 1; i < count; i++)
                if (temp > spot[i].dist) {
                        temp= spot[i].dist;
                        hit= i;
                }
        node[n].x= spot[hit].p.x;
        node[n].y= spot[hit].p.y;
        obst= spot[hit].oid;
        edge= spot[hit].lid;
        if (show) printf("obst#%d, edge#%d, node#%d0, obst,edge,n);

        pathclear= FALSE;
        Drawnode(n);
} else {
```

```
                pathcomplete= TRUE;
                printf("PATH COMPLETE0);
                node[n].x= loc[1].x;
                node[n].y= loc[1].y;
                Trace(n);
        }

} /* Reach */

/**********************************************************************/
Clear () {
/**********************************************************************/
        int l, j, count;
        struct coord h, t[50];

        if (pathcomplete) return;
        if (pathclear) {
                printf("Clear done.  Try Reach0);
                return;
        }
        l= upper? Next(edge) : edge;
        n++;
        if (show) {
        Repaint();
                printf(upper?"Upper direction0:"Lower direction0);
                printf("0lear: obst#%d, edge#%d, node#%d, ", obst,edge,n);
                printf("vertices: %d0, obj[obst].v_no);
                printf("First edge:%d, ", l);
        }
        pathclear= FALSE;
        node[n].x= obj[obst].v[l].x;
        node[n].y= obj[obst].v[l].y;

        while (!pathclear) {
                count= 0;
                for (j= 0; j < obj[obst].v_no; j++)
                        if (Cross(&h,&node[n],&loc[1],
                                &obj[obst].v[j],&obj[obst].v[j+1])) {
                                        t[count].x= h.x;
                                        t[count].y= h.y;
                                        count++;
                        }
                if (show) printf("Cross: %d0, count);
                for (j= 0; j < count; j++)
                        if (sqrt(pow(node[n].x - t[j].x, 2) +
                                pow(node[n].y - t[j].y, 2) ) > 1.0) count= 3;
                if (show) printf("Adjusted to: %d0, count);
                if (count > 2) {
```

```
                n++;
                l= Next(l);
                if (show) printf("Next edge:%d, ", l);
                node[n].x= obj[obst].v[l].x;
                node[n].y= obj[obst].v[l].y;
                if (show) printf("node#%d, ", n);
                Drawnode(n);
            } else {
                pathclear= TRUE;
                if (show) printf("Path clear @ node#%d0,n);
            }
        }
        Drawnode(n);
} /* Clear */

/**********************************************************************/
Next(vertex)
/**********************************************************************/
int vertex;
{
        if (upper)
                return (vertex < (obj[obst].v_no -1))? vertex+1 : 0;
        else
                return (vertex > 0)? vertex-1 : obj[obst].v_no - 1;

} /* Next */

/**********************************************************************/
Trace () {
/**********************************************************************/
        int i;

        Repaint();
        Drawnode(0);
        for (i= 0; i < n; i++) {
                _lineto(node[i+1].x, node[i+1].y);
        }
        printf("Number of nodes:%d0, n+1);
        printf(upper?"Upper direction0:"Lower direction0);
        Drawnode(n);

} /* Trace */

/**********************************************************************/
Drawnode (afew)
/**********************************************************************/
int afew;
{
```

```
        int i;
        struct coord h;
        for (i=0; i <= afew; i++) { /*generic--for Reach and Clear */
                h.x= node[i].x;
                h.y= node[i].y;
                _moveto(h.x, h.y);
                _setcolor(LTCYA);
                _setpixel(h.x, h.y);
                _ellipse( _GBORDER, h.x -3, h.y -3, h.x +3, h.y +3);
        }
} /* Drawnode */

/**********************************************************************/
Zip () {
/**********************************************************************/
        int i, j;
        int hitcount;
        struct coord h;

        hitcount= 0;
        _setcolor(LTBLU);
        _moveto(node[n].x, node[n].y);
        _lineto(loc[1].x, loc[1].y);
        for (i= 0; i < num; i++)
                for (j= 0; j < obj[i].v_no; j++) {
                        if (Cross(&h,&node[n],&loc[1],&obj[i].v[j],&obj[i].v[j+1])){
                                hitcount++;
                                _moveto(h.x, h.y);
                                _setcolor(LTMAG);
                                _setpixel(h.x, h.y);
                                _ellipse( _GBORDER, h.x -3, h.y -3, h.x +3, h.y +3);
                        }
                }
        if ((hitcount % 2) != 0)
                printf(" No solution: Different regions0);
                printf("(%d intersections)0, hitcount);
} /* Zip */
```

An Interactive Environment to Investigate
Robot Path Planning in a 2D Work-space

by

Khanh N. Hoang

B.S., University of Lowell, 1983
M.S., University of Lowell, 1986

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1989