C663 - Computer Vision - Final Project

Prof.: Andrew A. Hanson

The Simulation of a 6 legged autonomous robot guided by vision.

by Paulo W. C. Maciel

December 2nd, 1992

1. Introduction

In the past 20 years many simple autonomous robots have been built in prestigious institutions like MIT and CMU. Many of these mechanical creatures are guided by several different kinds of sensors, that infrared proximity sensors to sonars and ultimatelly by vision systems.

Problems with these kinds of sensors range from the variation of sensitivity between 2 infrared sensors or the angle of a sonar beam, to properties of objects like size and surface albedo. Specifically, with respect to vision the problem is even more difficult. A simple task that is very trivial for human beings like color discrimination is simply an open research problem[1]. The situation for vision systems get even worst when we are trying to analyze a sequence of imagens taken from a robot with a head-mounted camera. The images are simply too noisy.

As all of these problems suggest, we are not going to find solutions to all problems faced by the design and construction of real autonomous robots by trying to simulate these agents in a computer. Nevertheless, a good simulation should enable us to study aspects of the problem that could then be tested and improved in real machines.

The objective of this work is the simulation using computer graphics of a 6 legged autonomous robot which wanders inside a maze solely guided by what it sees from a built-in camera located in its "forehead".

In the next section, we state the navigational problem and show how, based on what the robot sees, it can find its way thru the maze without bumping into the maze walls.

The simulation section, describes how the simulation was created.

Then we describe how vision information was extracted from the image seen by the robot and the vision algorithms used to determine: when the robot is too close to a wall and needs to stop walking and initiate a turn, how it recognizes when it sees/don't see a corner, when it should stop a turn, and also while heading towards a wall when it needs to adjust its course.

Finally, we present the conclusion about the experiment and give directions for further experiments, pointing out some of the possible applications of a simulation of this kind if we allow more realistic environments.

2. Definition of the problem

The task of this simulation is, given a legged robot capable of moving forward and make right and left turns and a maze composed of a set of walls organized in such a way that each adjacent wall form a 90 degree angle with each other, how to make the robot travel inside the maze, without bumping into the walls, relying only upon what it sees from a head-mounted camera.

In this simple scenario, there are only a few possible kinds of images that the robot vision system, after runing an edge detector thorough the image, will capture:

At the point where 2 walls meet, together with the floor, we have a fork(like a 'Y').

Inside a corridor, the image seen is a set of 4 lines in perspective, going towards a square which corresponds to the end of the corridor.

At the point of a corridor where the left or right wall ends, together with the floor, we see an 'L'.

When facing a wall the image seen is composed of 2 horizontal lines.

Based on that, we describe the rules that are used to guide the robot based on the processed image of what it sees.

(a) Determining the distance to a wall.

While walking towards a wall, the processed image shows 2 horizontal lines that move apart, the bottom one going down and the top one going up. At each step, this image is analyzed and the distance from the bottom of the window that contains the processed image to the bottom horizontal line is measured. When this distance falls below a threshold, that is stablished in the program, the robot "knows" that it is too close to the wall and it needs to stop.

The robots field of view, is very sensitive to the distance it decides to stop before hitting a wall. If it stops too close, it will only see the wall and nothing else and will not be able to see a corner to the right or to the left. On the other hand, if it stops too far it may see, more complicated scenes formed by the way corridors are organized in the maze and might not be able to "understand" what it sees. Also, the behavior of the robot would be too constrained by the threshold distance stablished in the program.

The solution to this problem was found by making the robot look to its right and to its left and search for corners, whenever it comes too close to a wall.

(b) Deciding which way to turn.

After stoping in front of a wall and looking to the left and to the right there are 4 possible actions that the robot can take:

. If it sees a corner to the left and it sees no corner to its right, the only way to go is to the right and the robot initiates a turn to the right.

. If it sees a corner to the right and there is no corner to the left, it initiates a turn to the left.

. If there is one corner to the left and one to the right, its a dead end, and since the robot doesn't backup it stops.

. If there are no corners either to the left or to the right, both ways are equally good, therefore a function generates a random turn either to the left or to the right, and the robots start turning according to the chosen direction.

(c) Deciding when to stop turning.

Having initiated a turn, the robot monitors what it sees and decides when to stop turning. As the robot start turning, the 2 two parallel lines that were seen when the robot decided to stop(too close to the wall), become slanted and converging towards a point in the image. While seeing these 2 slanted lines the robot keeps turning. Eventually it will start seeing the end of the corridor and when it is oriented in a direction perpendicular to the wall at the end of the corridor it stops. .

The monitoring of the turn is done by selecting the mid-point of the processed image, which corresponds to where the camera is located(since this image shows the center portion of the camera image), and drawing a vertical line that intersects the walls edge. When the angle formed by the this vertical line on the center of the processed image with the line that determines the edge between the wall and the floor becomes 90 degrees the robot stops the rotation and starts walking again.

(d) Deciding when/how to adjust course.

After making a turn, the line corresponding to the edge between the wall at the end of the corridor and the floor and the vertical line from the center of the window are perpendicular. However, as the robot walks this changes and we don't have a horizontal edge anymore. The edge becomes discontinuous to the right or to the left by two or more pixels depending on the degree of disallingment between the robots heading and the wall at the end of the corridor. Since while walking down a corridor the robot doesn't check its distance to the right/left walls, if it is not heading perpendicular to the end wall, depending on the size of the corridor, it will eventually hit the lateral wall. Therefore, a course correction is needed.

Course correction is performed at each of the robots forward steps. At each step forward the vertical line from the center of the window needs to form a 'T' with the bottom edge of the corridors end wall.

This horizontal line(top of the 'T') is monitored. Whenver it becomes discontinued to the right the robot stops and initiates a turn to the left until the horizontal line becomes continuos again. Likewise, if the discontinuity is to the left, it generates a turn to the right.

After the course correcting turns, the horizontal line is continuous again and the robot resumes its normal course forward.

3. The Simulation

This simulation was done in a Silicon Graphics runing Iris and made use of the GL(Graphics library) functions which provides z-buffering and perspective drawing by specifying a look from and a look at point together with the near and far clipping planes.

Animation is accomplished by a loop that whenever something changes on the scene, like, the front leg moved forward, the whole scene is completely redrawn. The drawn is done to a "Backbuffer" frame buffer which is then copied to the "Frontbuffer" to be displayed while another scene is being again drawn in the backbuffer.

The robot is assembled by putting together 13 stretched cubes and one sphere to simulate the robots eyes and the floor is just a loop that draws squares on the xz plane.

The maze is drawn by a series of rotations and translations from the origin of the coordinate system of walls of 3 different sizes: small, medium and large, and is read from a file. The line below specify one of the walls of the maze:

5 m 90 -3 0 3 0 0 200

This describes wall number 5, which is specified by a 90 degree rotation of a medium size wall which is then translated to coordinate (-3,0,3). The rgb wall color is blue. The walls(s,m and l) are defined to be at the origin and are parallel to the xy plane. Therefore, different maze files enables different simulations.

The simulation provides 3 different windows. A large window displays the maze, the floor and the robot walking on it. A second window(camera view) is drawn taking the viewpoint to be the position of the robots eye and therefore displays what the robot is actually "seeing". A third window, contains a processed version of part of the camera view. This portion is taken to be the central part of the camera view image when the robot is heading towards a wall and taken to be the right(or left) portion of the camera view when the robot stopped in front of a wall and is looking right(or left).

This processed image is a black and white version of the camera view after the corresponding part of this image is passed thru a derivative operator which extracts the edges on the scene.

4. The Vision Algorithms

Before the vision algorithms can be applied, the edges on the camera view window need to be detected and drawn onto the processed image window. These edges are obtained in 2 steps:

1. The vertical and horizontal discrete derivatives of the image are obtained and converted to contain only black and white pixels. The derivative is obtained by making pixel(i,j) = pixel(i + 1,j) - pixel(i - 1,j) for the horizontal derivative and pixel(i,j) = pixel(i, j + 1) - pixel(i, j - 1), for the vertical derivative.

2. These two derivatives are then "ored" to obtain an image with both horizontal and vertical edges.

For every step forward or rotation performed by the robot, the correspondent part of the camera view is read into an array in memory, which is then processed in order to detect the edges on the image and is written back to the processed image window. This image is then ready for recognition by the vision algorithms. There are basically 3 algorithms that are used to:

1. Detect if the lowest horizontal line fall bellow a
threshold in order to determine the distance to a wall.

2. Detect a fork(similar to an upsidedown 'Y').

3. Check if the vertical line starting at the mid-point of the
processed image window intercepts a straight horizontal line.

4. Check for discontinuity of the the horizontal line, intercepted by
the vertical line from the mid-point of the processed image window

With the exception of the 4th algorithm, these algorithms return 'yes' or 'no', like 'yes' there is a fork in this image or 'no' there is not. The fourth algorithm also returns the side in which a discontinuity was found and is just a minor variation of the third algorithm.

(a) Algorithm for checking threshold

This is the most straightforward of the 3 algorithms and consists of a loop that checks each pixel on a vertical line starting at the midpoint of the bottom of the processed image array and goes until it reaches a black pixel. It then compares the vertical index of this pixel in the image array with the threshold. If it is greater then the threshold the robot has to stop walking otherwise it continues until the threshold is reached.

(b) Algorithm to detect a fork.

The way this algorithm works is basically the following: Start at the bottom midpoint of the processed image array and go until it finds a line. Follow the pixels in this

line until it finds a fork or either it finds a vertical line(corresponding to the upper part of an inverted 'Y') without the other branch of the inverted 'Y', or the indices of the array fall off the array. The algorithm is described below in more detail:

```
input: The array containing the processed image
output: 0 - no fork is found, 1 - found a fork.

Fork Detection Algorithm
{
    Select (mi,0) to be the midpoint of the bottom of the array.

    while(has not reached a line) {
         get the next vertical pixel.
         if (there are no pixels left)
            return(0);
   }

   /* Start searching for the fork. */
   While(there are pixel in the line to process) {

         /* Here the neighbor means the one to the right or to the up
            right of the pixel. */
         Set (i,j) to be the neighbor of (i,j) in the line.

         /* Check for a vertical line starting at this pixel. */
         if (there is a vertical line at (i,j)) {
            /* Check the right neighbors of (i,j) */
             if (there are neighbor pixels to the right of (i,j))
                 /* Found a fork. *.
                 return(1);
             else
                 /* No fork is found. */
                 return(0);
        }
    }
   return(0);
}
```

(c) Intersection with a straight line.

In the same way as in the previous algorithms a vertical line is followed from the bottom mid-point of the image array until it reaches a line which is then checked to see if it is a "straight line" or not.

Since the purpose of this algorithm is basically to determine when the robot should stop turning what is meant here by a straight line is one that doesn't have discontinuities

neither to the right nor to the left of the intersection with the previously mentioned vertical line.

Therefore, from this intersection point we follow the pixels to its right(and to its left) until we either: find a discontinuity, find an 'L' or fall off the image array. If we found either an 'L' or the index felt out of the image array in both right and left side of the line, then it is considered to be a straignt line. Otherwise, it is not. The algorithm is detailed below:

```
input: The array containing the processed image
output: 0 - no straight line found, 1 - found a straight line

Straight Line Check Algorithm
{
    Select (mi,0) to be the midpoint of the bottom of the array.

    while(has not reached a line) {
         get the next vertical pixel.
         if (there are no pixels left)
            return(0);
   }

   Set (i,j) to (mi,0);
   /* First, check the right side of the intersection point. */
   While(hasn't reached the right limit of the image array) {

         /* Check for a vertical line starting at this pixel. */
         if (there is a vertical line at (i,j)) {
            /* Found an 'L'. *.
            exit loop;
        }

         /* Here the neighbor means the one 'STRICTLY'
            to the right of the pixel. */
         Set (i,j) to be the neighbor of (i,j) in the line.

         if (there are no pixels to the right)
             /* found a discontinuity to the right. */
             return(0);
   }

    Set (i,j) to (mi,0);
    /* Then, check the left side of the intersection point. */
    While(hasn't reached the left limit of the image array) {

         /* Check for a vertical line starting at this pixel. */
```

```
            if (there is a vertical line at (i,j)) {
                /* Found an 'L'.  The line is a straight line. */
                return(1);
            }
            /* Here the neighbor means the one 'STRICTLY'
               to the left of the pixel. */

            Set (i,j) to be the neighbor of (i,j) in the line.

            if (there are no pixels to the left)
                /* found a discontinuity to the left. */
                return(0);
        }
        return(1);
    }
```

(d) Line Continuity check algorithm.

This algorithm is responsible for determining when the robot should adjust its course. If there is a discontinuity on the line which is the bottom edge of a wall the algorithm returns 0 and also indicates the side of the discontinuity, either left or right, so that the robot can correct its course by making a turn in the opposite direction. The algorithm is the same as the above with the only difference being that it records the side in which the discontinuity occured. Therefore the output of this algorithm is:

```
output: 0 - no straight line found, 1 - found a straight line and the
        SIDE of the discontinuity, either LEFT or RIGHT.
```

and the line in the previous algorithm that says:

```
            if (there are no pixels to the right/left)
                /* found a discontinuity to the right/lef. */
                return(0);
```

is changed to:

```
            if (there are no pixels to the right/left)
                /* found a discontinuity to the right/lef. */
                set SIDE to RIGHT/LEFT.
                return(0);
```

5. Conclusions and Extensions to this project

What we've described in this work is a walking robot that is able to interpret what it sees and make navigational decisions based solely upon that interpretation and find its way through a simple maze.

The simulation perfectly shows that its judgements were indeed correct and when the experiment was run with a 22 wall maze it was able to walk, make correct turns and for 2 times, after walking a few steps right after a turn, it was able to adjust its course and headed towards the next wall.

However, this experiment is still very limited with respect to the orientation of the maze walls(as it is, 2 adjacent walls need to form a 90 degree angle) and to the realism of the scenes it sees.

In a more realistic experiment, we would like the robot to wander inside the maze(possibly an office or an industrial plant), identifying landmarks which could be walls with certain colors, textures or drawings, 3D objects along the way and build an internal representation of the terrain covered together with the location of the seen objects. Ultimatelly, what we would like, is to be able to tell the robot, things like: "Go pick up the metallic bar in front of the wooden wall".

Clearly, this is not a simple task.

Some of the features that would be needed to accomplish such a task are:

. Build a representation of the seen object, possibly using techniques that recover shape from shading.

. Use of matching techniques that would interpret the acquired information as being similar to a generalized cone representation of a pre-stored known object.

. Use techniques like the Hough transform to identify, patterns on walls, or the outline of objects.

. Make use of techniques like stereopsis to identify, how long a corridor might be.

The possibilities are inumerous. Although, simulations like the one described are not likely to provide solutions to real physical problems, it provides a platform in which many ideas can be tested.

References:

[1] "The role of learning in autonomous robots",Rodney A. Brooks, MIT AI Lab.

[2] "Dynamic simulation of autonomous legged motion", McKenna, MIT Media Lab, in Siggraph 90.

[3] Graphics Library Programming Guide

[4] Graphics Library Windowing and Font Library Programming Guide

[5] Graphics Library Programming tools and techniques