18-551 Spring 2005
Group 6 Final Report
EZ Park

Paul Li cpli@andrew.cmu.edu
Ivan Ng civan@andrew.cmu.edu
Victoria Chen vchen@andrew.cmu.edu

Table of Content

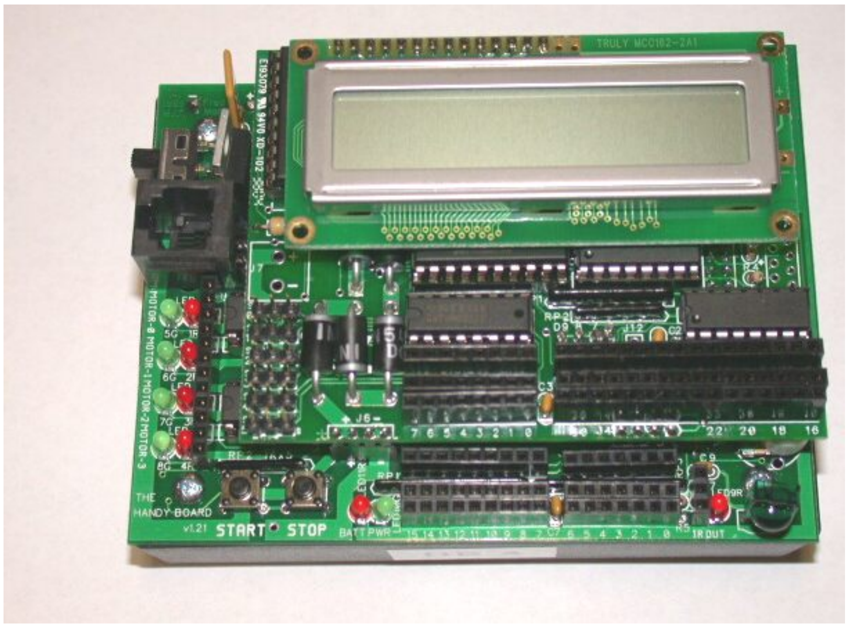
Introduction

Problem

Drivers generally have a harder time doing back in parking as oppose to head in parking. What makes back in parking difficult is that we do not have the full vision behind the vehicle. Also, it is hard to estimate the space on the side of the car to ensure enough room on both sides for the driver and passengers to get in or out of the car easily.

Solution

We implemented a system using a LEGO© car with Logitech QuickCam® Zoom™ - Silver mounted on it to simulate the situation which there is a camera behind the vehicle. To solve the abovementioned problem, we first identify the white grid lines in each frame. By calculating the tilted angle between the lines and the direction of the automobile, we know how much steering is needed and how far the car is from the parking space at that moment. The LEGO car is controlled by Handyboard as shown below.



Previous Works

A similar previous project was 2004's group 4, "Car Eye for the Drunk Guy". Their project used Hough Transform to recognize the white lines on highway. They then processed the video from a car-mounted camera to keep track of the location of road lines and determined whether the driver is driving safely.

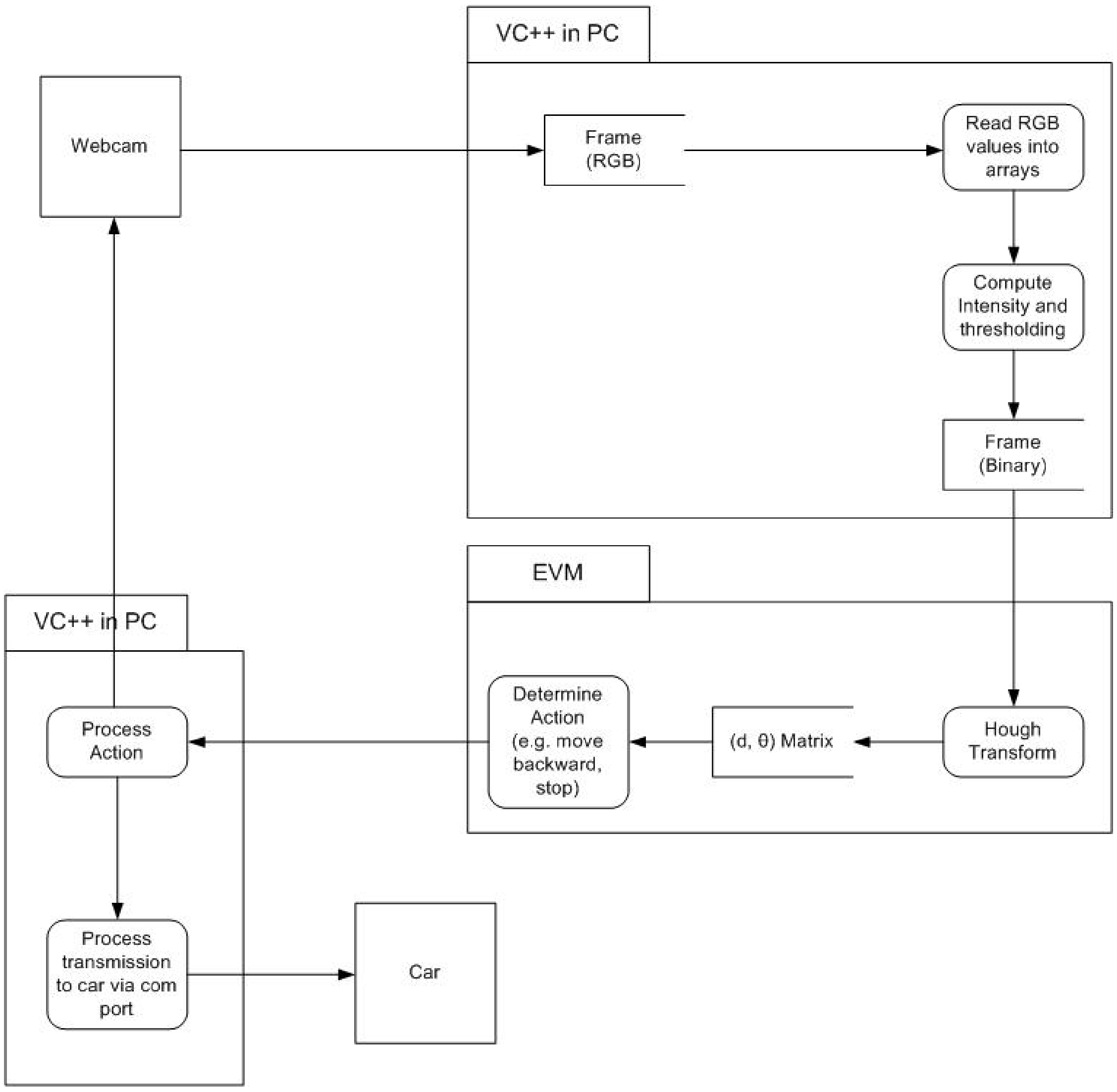
There are certain differences between their project and ours. Our project is *real-time* and we need to recognize three parking lines of different angles. On top of that, we need to program the Handyboard to recognize instruction from the PC and perform the action we desired.

Assumptions

Certain assumptions were made in our project:

1. For our program to work properly, it is necessary to ensure that the mounted-webcam captures all three lines of the parking space.
2. We assume that the model car is in a position such that only one turn is necessary for parking. Backing out and making two turns are out of the scope in this project.
3. The model needs to be either on the imaginary middle line of the parking space or some angle away from it.
4. All calculations and calibrations are assumed to be perfect.

System Overview



VC++ in PC
Webcam
Frame (RGB)
Read RGB values into arrays
Compute Intensity and thresholding
Frame (Binary)
EVM
Hough Transform
(d, θ) Matrix
Determine Action (e.g. move backward, stop)
VC++ in PC
Process Action
Process transmission to car via com port
Car

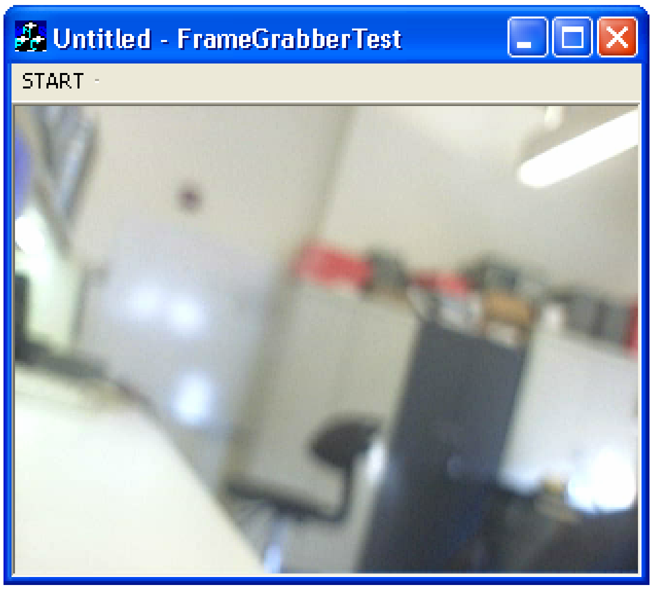
The system starts after the user clicked “Start” on the PC program. The webcam on the car will capture a frame and convert the RGB image to grayscale image and then perform thresholding to obtain a binary image. The binary image will be sent to the EVM. Straight lines are detected and the appropriate action is determined. The action is sent back to the PC where the PC will send the action to the car. After the car is done performing the instruction, the webcam will capture another frame and the same process repeat.

Detailed Algorithm

Framing

C++ code for framing is obtained from http://www.codeproject.com/audio/avicapwrp.asp. The source codes, Frame Grabber, offers options for formatting the webcam, such as resolution, colors, brightness, contrast, etc. This program captures frame from the webcam and saves the frame as a bitmap file any time when the "Save As" button is clicked. However, we did not use these options, and we hardcode the image size to be 160 * 120.

The modified window for Frame Grabber looks like this:



The entire parking process is automated once the START on the toolbar is clicked.

We only need to capture two frames for the entire process. The first frame is taken at the starting position, as shown in Fig. 1. The car would then back up straight a little and turn until the car is parallel with the parking lines. The second frame will be taken once the car is stopped and parallel to the parking lines, as shown in Fig. 3.

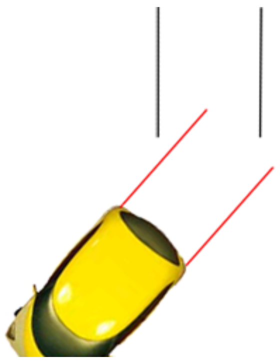


Fig. 1

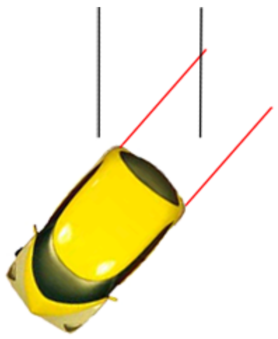


Fig. 2

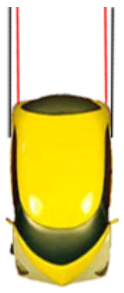
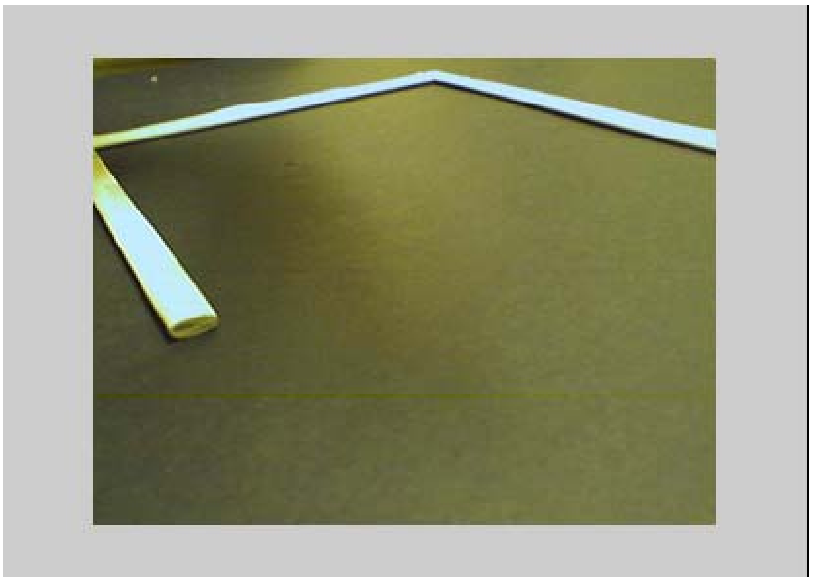


Fig. 3

RGB Image

The reference we used is *http://www.microsoft.com/msj/1097/wicked1097.aspx*. The program generates a color palette of a bitmap file. We found the CQuantizer class particularly useful. It helps to convert the 32-bit DIB to 24-bit RGB format one scan line at a time. Then it reads individual pixels by scanning the line from left to right using the same logic it uses to read 24-bit DIBs. We modified the codes such that when it reads individual pixels, it saves the RGB values into three arrays, redA, greenA and blueA at the same time.



RGB Image

Grayscale

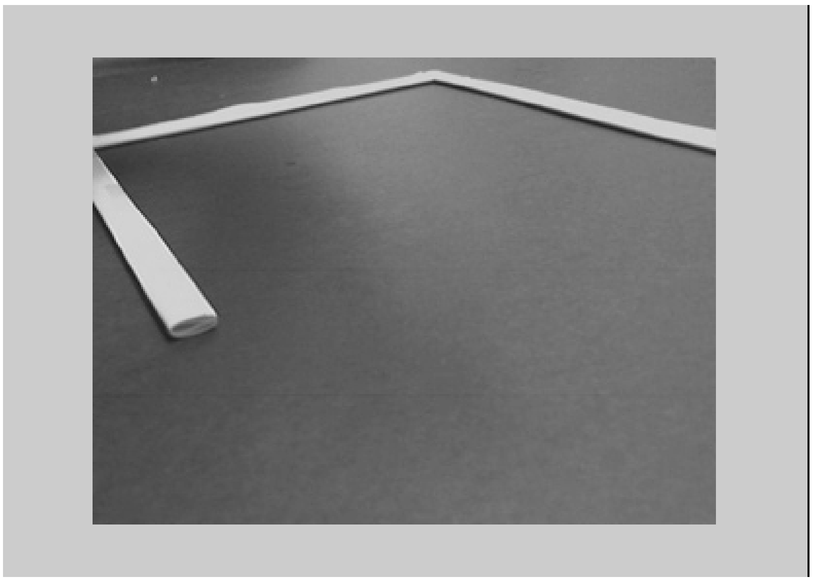
We then grayscale the RGB values of each pixel according the following equation:

$$gray = 0.257 \times red + 0.504 \times green + 0.098 \times blue + 16$$

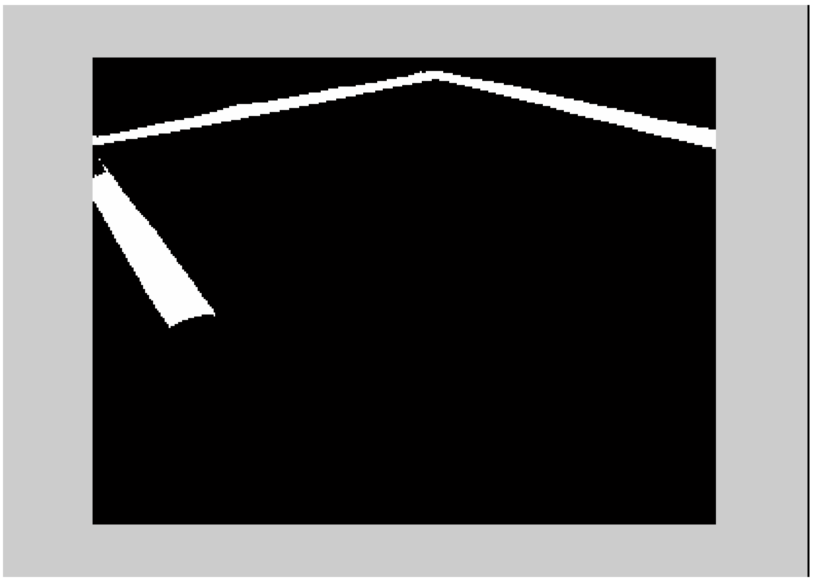
The maximum and minimum of these gray values are recorded to calculate the threshold by using the following equation:

$$threshold = 0.7 \times (\max - \min) + \min;$$

After that, we need to convert the grayscale image to binary image. If the gray value of a pixel is greater than the threshold value, we assign a "1" to it to represent an "ON" pixel, vice versa. These binary numbers are saved in an array and the array is sent to EVM for further processing.



Grayscale Image



Binary Image

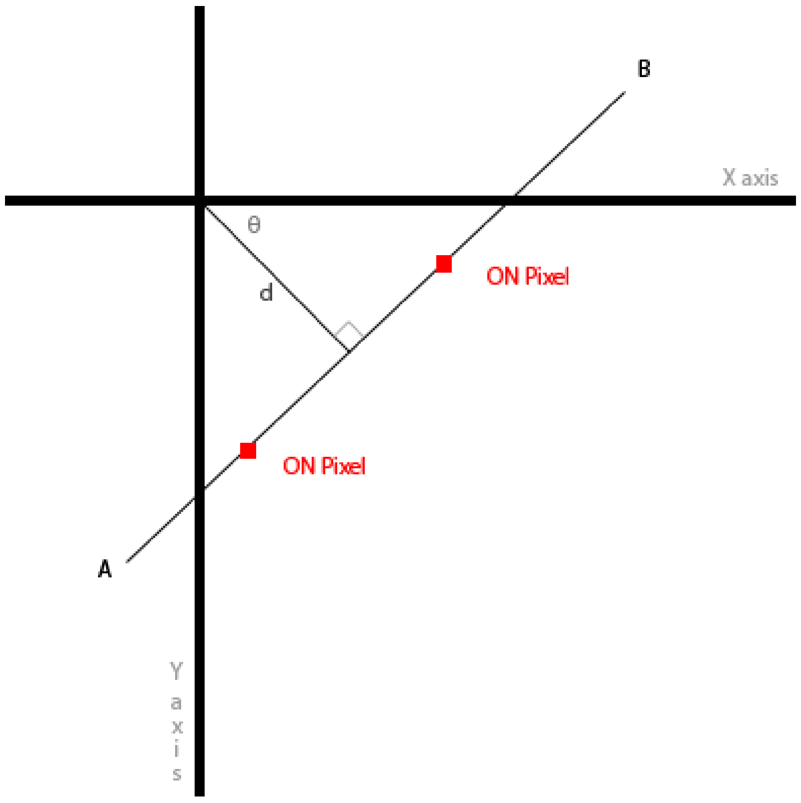
EVM Input and Output

The input to the EVM is a 160 * 120 pixels binary image. Since the EVM is capable of receiving 3MB/sec, the transfer of image data from the PC to the EVM is not a problem. The output of the EVM is the instructions to the Handyboard. They are numbers which represent the desired movement of the LEGO car.

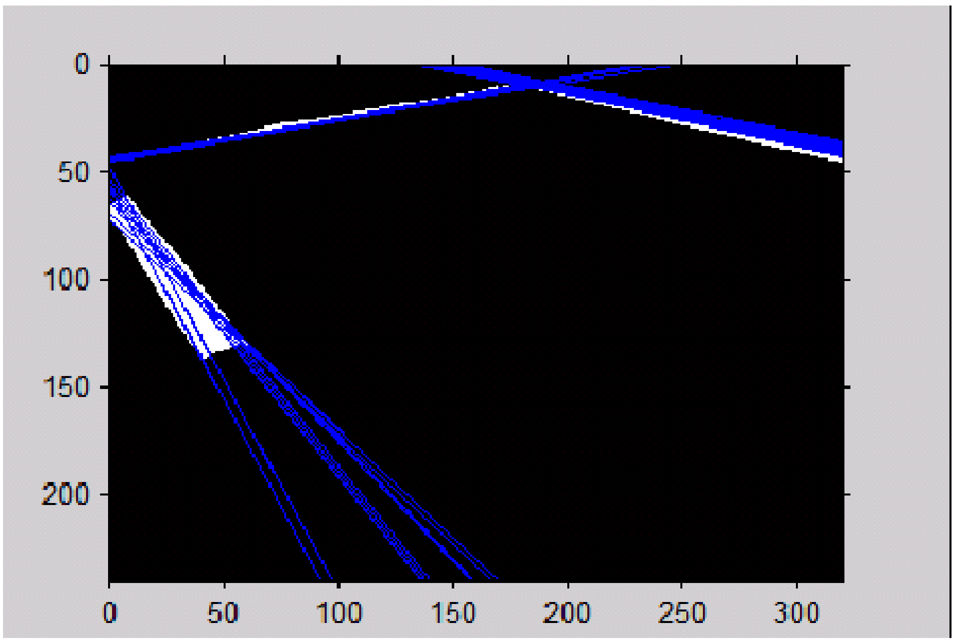
Line Detection

To detect the straight lines from the captured frame, we used the following method.

```
for every 'ON' pixel
    for all the other 'ON' pixel
        draw a line AB to join the two 'ON' pixels
            determine the $\theta$ and d of the line
            increment the *Accumulator* array of the cell (d, $\theta$)
```

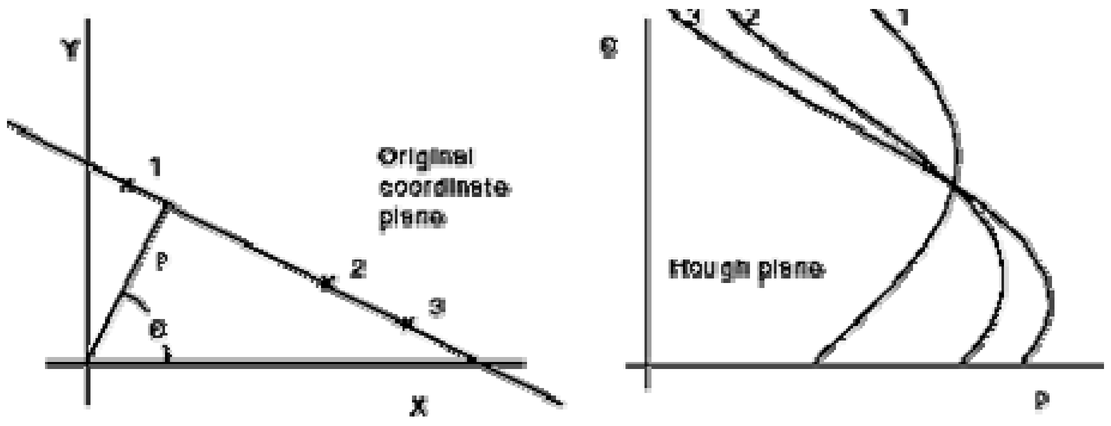


From the *Accumulator* array, choose (d, θ) points which have a value higher than a threshold value and classify them as “lines”.



Detected lines are plotted as blue lines in the figure above.

There is a popular line detection algorithm called *Hough Transform* that we did not use. Hough Transform transforms the image from the Original Coordinate Plane to the Hough Plane. Lines are parameterized according to the equation $p = x * \cos \theta + y * \sin \theta$. *Hough Transform* runs much faster but lines of 90^o can not be detected.

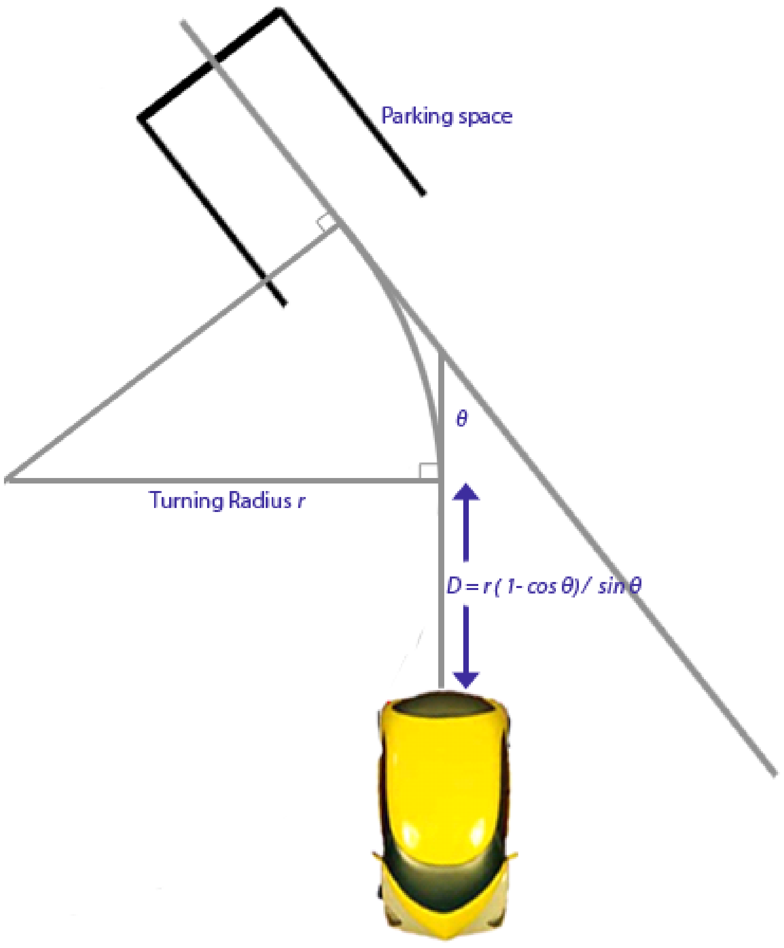


Line Clustering

To cluster the many detected lines into 3 groups, we put lines of distance $(d_1\text{-}d_2)^2 + (\theta_1\text{-}\theta_2)^2$ less than a certain threshold into the same group. After clustering, an average is calculated to represent each line. In most of the cases, this method works well and do not require a lot of calculation. Sometimes, however, a 4th group of line might be found and affect our detection algorithm.

Another method of finding three groups of lines was suggested during the demo. Instead of using a fixed threshold to find out the lines, start out with a high threshold and lower the threshold until three groups of lines are detected.

Back in Calculation



From the starting position, the car first back up straight for distance *D.* The car will then turn with a turning radius of *r* and travel a distance of *r*θ*. After this, the car should be inside and parallel to the parking space.

Serial Port Communication

Serial port is used for data transfer between PC and Handyboard.

Handyboard

The Handyboard can read and write character (ASCII 1 to 127) to the serial port.

PC

A serial library in C++ is available at *http://www.codeproject.com/system/serial.asp*. It provides the *send* and *receive* functions to send the instruction to the handyboard.

Instructions for the Movement of the Car

The Visual C++ program sends the following information to the Handyboard via Serial Port

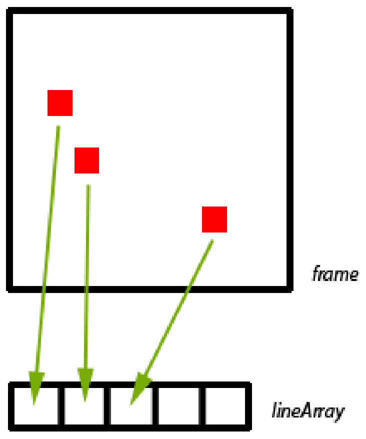
(1) Back up distance
(2) Left or right indicator
(3) Left or right distance

The distance (the time the motor has to be ON) is represented by values from 1 to 126 and the Left/Right indicator is either 1 or 2.

After the Handyboard received an instruction, it writes the number 127 to the serial port so that the Visual C++ program can read and proceed to send the next number.

Speed Issues

We have to access 'ON' pixels many times when we detect the straight lines. Instead of searching through the whole image every time, we first find all the 'ON' pixels in the image and store the 'ON' pixel array index into another array lineArray. Now we only need to access lineArray when we perform straight line detection.



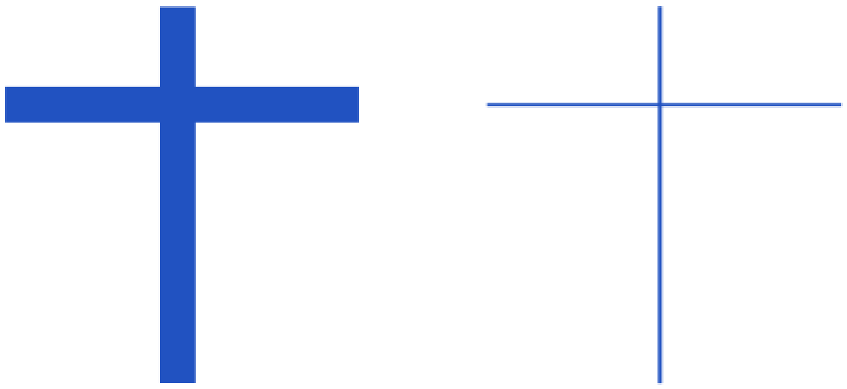
The size of *lineArray* depends on the number of 'ON' pixel in the captured frame. On average there are around 800 'ON' pixels in a 19200 pixel frame. Therefore, the size of *lineArray* is much smaller than that of the frame.

However, our line detection algorithm requires a lot of calculation and needs a relatively long time to finish the whole process. The time needed to complete the line detection also depends on the number of 'ON' pixels. A frame with 800 'ON' pixels would take roughly 10 seconds to finish the straight line detection process.

During the demo, we were suggested to perform *Skeletonization* on the image before line detection. *Skeletonization* is the process of peeling off of a pattern as many pixels as possible without affecting the general shape of the pattern. In other words, after pixels have been peeled off, the pattern should still be recognized. The skeleton hence obtained must have the following properties:

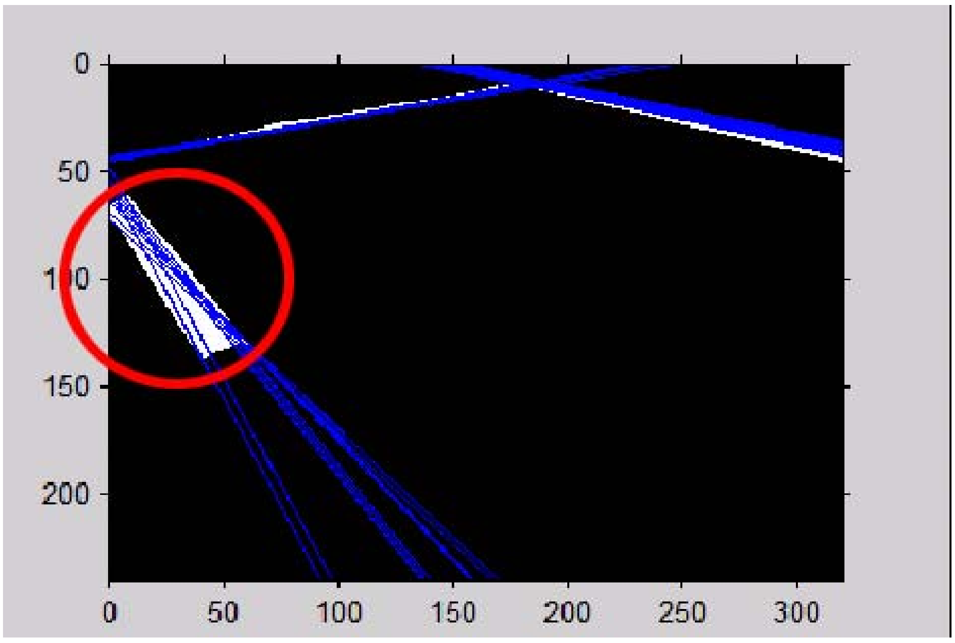
- As thin as possible
- Connected
- Centered

When these properties are satisfied, the algorithm must stop.



Before *Skeletonization* After *Skeletonization*

After *Skeletonization*, the number of pixels that need to be processed would decrease drastically and thus, increase the speed of our system. Also, we will not get multiple lines of slightly different *d* and *θ* from one thick line as shown in the figure below.



Errors

Motor Calibrations

Major errors come from calibrations. We supply the model car with ±100V to its motor for forward and backward motion, and also another motor for left and right turn. To control the movement, power is supplied to each motor for a certain period of time. We measured the time versus distance traveled for all four motions: forward, backward, left turn, right turn. All of these motions are not linear with respect to time due to the fact that there is a sudden surge of power for its initial movement, thus making the first few 0.1 seconds more erroneous. Also, due to the unsteadiness of the motor, the model car does not travel the same distance every time, given the same amount of power and time.

Starting Position

When the car's starting position is just in front of the parking space and parallel to the parking case, the lines detected are very close to 0^o and 90^o. We have a special instruction for this case. However, the line detection algorithm is not perfect. Errors often occur when the car is almost parallel to the parking space and caused the system to think that the car is parallel to the parking space. In this case, our system will fail.

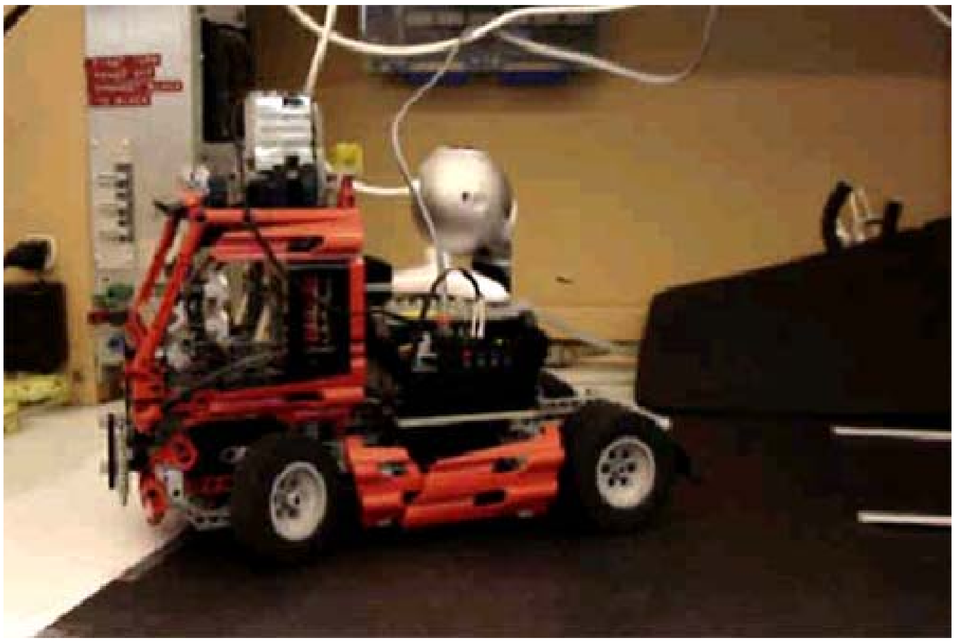
Webcam Position

Since the webcam is not LEGO pieces, it can not be fixed onto the car tightly. The position of the webcam could be moved and could produce error in our calculation. A slight movement of the webcam could make the car unable to move to a position parallel to the parking space and such.

Demo

During the demo, we put the car at a distance from the parking space and let our system control the movement of the car. In most starting positions, with or without obstacles blocking part of the white lines, our system worked. Starting positions where the car is almost parallel to the parking space or only a very little portion of one of the lines can be seen caused our system to fail.

Initial position – facing the parking space


Back up straight and turn

Turn until the car is parallel to the parking space

Back up straight until the car reaches the end of the parking space

References

1. **C++ code for framing.**
   http://www.codeproject.com/audio/avicapwrp.asp

   The demo files provide an application to capture frames from webcam and save them as bitmap files in real time. There are other functions such as editing the settings of the webcam which is not used in our project.

2. **C++ code for creating palette of a given bitmap file.**

   http://www.microsoft.com/msj/1097/wicked1097.aspx

The source codes only pick out color from the bitmap file and convert it to a color palette. Our project, however, requires the RGB values of each pixel, thus changes are applied to the source codes to fit our needs.

3. **C++ code for serial communication.**
   http://www.codeproject.com/system/serial.asp

   The source files contain *Send* and *Receive* functions of the serial port, which is all that is needed for this part of the project.

4. **C code for serial port communication for the Handyboard**
   http://www.handyboard.com/software/contrib/drushel/