Lab 10 - Can Collection

Marcus Pham - 2018

Aim:

To implement a robot program that starts in the driving area then searches the environment for red cans and drives towards the red can (effectively collecting the can) and returning to the original position.

Equipment:

- Mobile Robot with Embedded Controller incl. sensors and motors
- User Laptop to program robot (Mac OS or Windows Preferred)
    - Have a simple text editor (XCode, TextEdit, Notepad)
    - Installed FTP file sharing program (eg. FileZilla)
    - Installed putty (for Windows)
    - Links below (Appendix 1)

1. Revision - Capture and display the image to the screen

- We will start by displaying an image and identifying the red can.
- To capture an image, we must first allocate sufficient space to store the incoming bytes from the camera. To do this, simple create an array of BYTEs to the size that we want to use (in this case, we are capturing a 160x120 image in RGB format, and hence will need 160x120x3 bytes of space to store the image). In the eyebot library the sizes are already defined and hence we can create the array using:
  BYTE img[QQVGA_SIZE];
- Create a menu to exit:
  LCDMenu(“”, “”, “”, “End”);
- We need to now initialise the camera and specify the resolution that we wish to capture
  CAMInit(QQVGA);
- We now can create a variable to specify if the end button has been pressed
  int exit = 0;
- In order to keep the program running until we wish to exit, we must create a loop, continuously capturing images and displaying them on the screen.

- int keycode = 0;
- **while(keycode!=KEY4) {**
  CAMGet(img); //captures the image and stores in the array
  LCDImage(img); // displays the image
  if(keycode == KEY1) {
  }
  keycode = KEYRead()
  }

- Finally, we should close the camera and exit the program
  CAMRelease();
  return 0;

2. Identify Red Can

- So now that we can capture an image and display to the LCD, we can do a bit of image processing to determine where the red can is.
The general algorithm to find the red can is:

1. First determine which pixels are 'red', we can do this from your algorithm from the previous lab… And then calculate the binary image… Remember to allocate space…

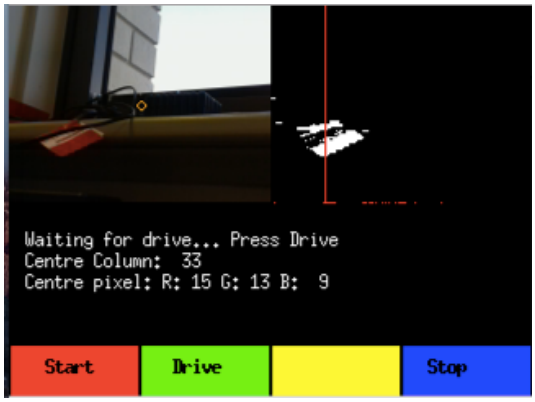
2. Now that we can a binary array, we can display this next to the colour image to confirm that it is indeed identifying red pixels. We can use LCDImageStart to specify the position of the new image and LCDImageBinary to display our binary image. We also should change back to the original positions for the original colour image to print to the left side:

   ```
   LCDImage(img);
   LCDImageStart(160, 0, QQVGA_X, QQVGA_Y);
   LCDImageBinary(binary);
   LCDImageStart(0,0, QQVGA_X, QQVGA_Y);
   ```

3. Now we can confirm and check that it is indeed identifying red. The next step is to make a histogram to identify the column where the most red pixels lie. We can later use this to centre our robot and drive towards the red can.

Additionally we should have some threshold to determine if there is not enough red to correctly say that the red can is in view:

Following this, we should see our red can a **printed statement to tell us which column where the most red pixels are and a vertical line on our binary image showing the column with the most red.**



3. Controlling the Car

Now that we have identified our red can, we can now attempt to drive towards the can.

- There are already built in drive functions to drive the car to the correct location, the main functions needed are:
  - **VWStraight(dist, speed); //drive straight**
  - **VWDriveWait(); // waits for completion of drive step**
  - **VWTurn(angle in degrees, speed); // turns on the spot**
  - **VWCurve(dist, angle in degrees, speed) //drives the car on a curve**
  - **VWDrive(x, y, speed) // also drives in a curve to location x (in front) y (to left (positive)/right (negative)**

- Also you will need to avoid the walls as well as determine the distance in front of the can. This is done by reading the values from the PSD laser sensors. The PSDs will output a integer value corresponding to the distance, the smaller the number, the greater the distance. The functions to read these are:
  - **PSDGetRaw(<psd_no>) // 1 for centre, 2 for left, 3 for right**

- To test the hardware, you can go to the hardware menu and select motors/encoders or PSDs to get the correct distances needed for your drive

The easiest method to ensure that the can in directly in front of the vehicle is to ensure the column with the most red lies within the centre third of the screen (keep in mind the image has a width of 160 pixels to do your calculation). Then we can turn the car accordingly to align the image with the front of the car. This must be done continuously until the car reaches the can:

4. Returning to the original position

- Now that we have reached the can, the final step of the process is to return to the starting position.

The robot has inbuilt encoders to record and keep track of the location of the car. To simplify matters, there are already some predefined functions to allow you to easily return to the starting position.

First step is to set the starting position before any driving to 0:

VWSetPosition(0,0,0);

Then at the end of our trip we can call VWGetPosition, then we can reverse the process to calculate the turn angle and distance to return.

```
int x, y, phi;
int dist;

 //Get the information to return
 VWGetPosition(&x, &y ,&phi);
```