Mike Facelle
Embedded Linux Project Documentation

OPENCV INSTALLATION

1)
I followed the steps at:
http://tech.enekochan.com/2012/05/21/install-opencv-2-3-1a-in-mac-os-x-10-6/
to install OpenCV.
Because of errors trying to run make, I had to download opencv 2.3.1, not the latest version. This utilized make and cmake. To install, the following commands were run:

```
cd OpenCV-2.3.1
mkdir build
cd build
cmake -G "Unix Makefiles" ..
make sudo
make install
```

This built and installed the libraries needed for OpenCV.

2)
In order to run OpenCV programs with XCode, I followed the steps at:
http://tech.enekochan.com/2012/05/21/use-opencv-in-xcode-4-for-a-mac-os-x-application/
The instructions said to add several folders and flags to specific build parameters in the "*Build Settings*" tab.

Inside "*Search Paths*":
  Header Search Paths: **`/usr/local/include`**
  Library Search Paths: **`/usr/local/lib`**
Inside "*Linking*":
  Other Linker Flags: **`-lopencv_core -lopencv_highgui -lopencv_imgproc`**

3)
I also installed OpenCV on the Raspberry Pi by following steps on their tutorial page:
http://docs.opencv.org/doc/tutorials/tutorials.html
This utilized cmake, as the Mac installation did. The command used were:

```
cd ~/opencv
mkdir release
cd release
cmake -D CMAKE_BUILD_TYPE=RELEASE -D CMAKE_INSTALL_PREFIX=/usr/local ..
```

and after entering the new temporary directory, the last two commands were entered:

```
make
sudo make install
```

This took quite a bit longer than the Mac OS installation, as expected. The openCV libraries were stored in `usr/local/` as specified by the cmake command.

OPENCV PROGRAMMING

4)
I followed tutorials at the OpenCV website:
http://docs.opencv.org/doc/tutorials/tutorials.html
To learn basic functionality, and more advanced techniques like line detection.
Line detection involved first using the Canny Edge method to reduce a black and white image to just lines that mark the borders between shades (when the change in color is sudden). Taken from my main.cpp file, the comments indicate parameters.

```
// source, destinaton, threshold1, threshold2, aperturesize=3
Canny(src, dst, 100, 100, 3);
```

Lines can then be detected using the HoughLines function. Taken from my main.cpp file, the comments indicate parameters.

```
// =============== PROBABILISTIC HOUGH LINE TRANSFORM =================
//      creates line segments
// dst: edge-detector output (should be grayscale)
// lines: vector to store lines found;
// rho: resolution of parameter r in pixels (using 1)
// theta: resolution of parameter theta in radians (using 1 degree)
// threshold: The minimum number of intersections to “detect” a line
// minLinLength: The minimum number of points that can form a line.
//     Lines with less than this number of points are disregarded.
// maxLineGap: The maximum gap between two points to be considered
//     in the same line.
HoughLinesP(dst, lines, 1, CV_PI/180, 25, 50, 20 );
```

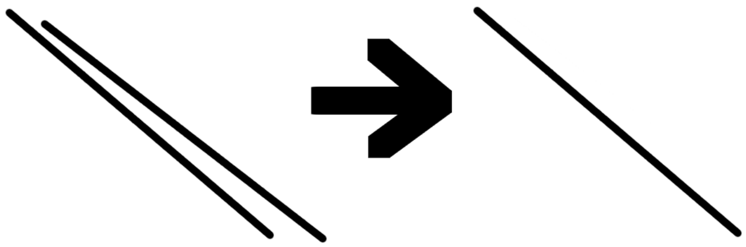
5)
I filtered out some of the horizontal lines, since street lanes will almost always be vertical, by determining their slope and removing them if it falls below a certain tolerance value (0.3 was used).
I then wanted to concatenate lines that are adjacent to eachother, or simply the same line, but broken up. This meant a line must meet the following criteria:

1) *Their slopes are equal (within a tolerance)*
2) *Their x-intercepts are equal (within a tolerance)*
   - x-intercepts used because the lines are mostly vertical

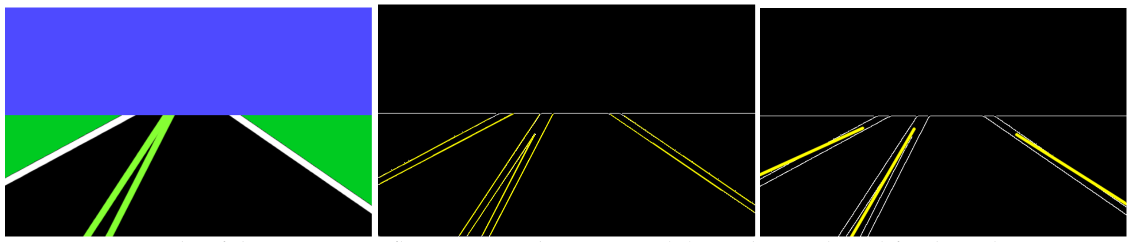
3a) *For adjacent lines: the nearest point on one line has a smaller magnitude than the farthest point on the closer line.*
3b) *For separated lines: the nearest point on one line has a larger magnitude than the farthest point on the closer line.*

A line, `l`, is deemed “closer” if the magnitude of its far point (`l[2]`, `l[3]` for the `Vec4i` format) is larger than the magnitude of another line’s far point (`l[0]`, `l[1]`). This check is only done if two lines are already deemed “the same,” so determining the nearest/farthest points are done by finding the nearest/farthest y-values.

When two lines are found to be “adjacent” or “separated” they are fixed by creating a new line.
If separated, then the nearest point of the new line is simply the smallest magnitude point of the two lines, and the farthest point of the new line is simply the largest magnitude point of the two lines.
If adjacent, then both points on the new line are the average of both the nearest and farthest points of the two lines.


Example of combining two “adjacent” lines

Due to the nature of how images are encoded in OpenCV, the point (0,0) is in the upper-lefthand corner, as opposed to the lower-lefthand corner of most x-y plots. This means that “max” y-values (from the point-of-view of the user) are actually smaller in magnitude than “min” y-values. This means that in the code, greater-than and less-than signs are swapped and calls to max() and min() are also swapped. It’s possible this was unnecessary, but it made it easier to think about the lines as though they were on a standard pair of axes.


Example of the program’s flow: Original image, With lines detected, and final product

Since lines may not actually reach the edge of an image, a function extend_lines() was used to push the lines to the end of the image. This works by checking x and y values and determining if they are near the image’s edges:
`0` *for the left edge,* `#columns-tolerance` *for the right edge, and* `#rows-tolerance` *for the bottom edge,* where tolerance is a variable, in pixels. The number of rows and columns can be found easily by using the variables `image.rows` and `image.cols`.

APPENDIX

7)

The following is the main body of the header file used for this project

```
// constants for opencv functions (canny, houghlinesp)
#define CANNY_T1        100
#define CANNY_T2        100
#define CANNY_APERTURE  3
#define HLINES_THRESH   25
#define HLINES_MINLINE  100
#define HLINES_MINGAP   20

// constants for referncing points (x1,y1)(x2,y2) = l1[0,1,2,3]
#define X1  0
#define Y1  1
#define X2  2
#define Y2  3

// tolerances for determining if two lines are "the same"
const double HORIZONTAL_TOLERANCE = 0.33;    // how far from slope=0 is considered horizontal
const double POINT_TOLERANCE = 100;     // how close (in px) for two lines to be "equal" (x-intercept)
const double SLOPE_TOLERANCE = 0.70;    // how close (in %) two lines slopes must be to be "equal"
// tolerance for how close to edge to extend to a side of image (in px):
const int NEAR_EDGE = 100;

// functions to reduce number of lines in image
// ---
void remove_horizontal(vector<Vec4i> *);    // removes horizontal lines from the vector<Vec4i>
void remove_lines(int,int,vector<Vec4i>*);  // removes two lines from a vector of lines
bool horizontal(Vec4i);                     // determines if a line is horizontal (below tolerance)
// ---
vector<Vec4i> combine_lines(vector<Vec4i>); // combines adjacent/seperated lines
vector<Vec4i> extend_lines(vector<Vec4i>,int,int);  // extends lines to reach end (bottom, edges) of
screen
// ---
bool greater_than(Vec4i, Vec4i);            // returns true if first line is higher up than second
void swap(Vec4i*, Vec4i*);                  // swaps two lines to pass to adjacent/seperated()
correctly
// ---
bool same_line(Vec4i, Vec4i);               // returns true if l1,l2 are the "same" line
bool adjacent(Vec4i, Vec4i);                // returns true if l1,l2 are adjacent
bool seperated(Vec4i, Vec4i);               // returns true if l1,l2 are seperated but the "same" line
// ---
double slope(Vec4i);                        // returns slope of the line passed
double y_intercept(Vec4i);                  // determine y-intercept of line passed
double x_intercept(Vec4i);                  // determine x-intercept of line passed
int mean(int,int);                          // returns mean between two points
```