Mike Facelle
Embedded Linux Project Documentation

OPENCV INSTALLATION

1) Installation on Mac

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) Running XCode with OpenCV

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) Intallation on the Pi

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) Detecting Lines using OpenCV

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) Reducing Lines to Only the Lane Lines

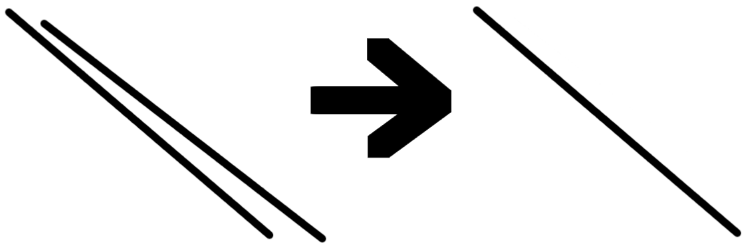
I filtered out some of the horizontal lines, since street lanes will almost always be vertical, by determining their slope and removing them if it falls below a certain tolerance value (0.3 was used).
The lines which are “in the sky” (above half of the image) were also filtered out, since it is incredibly unlikely that street lanes will be found in the upper half of the image.
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

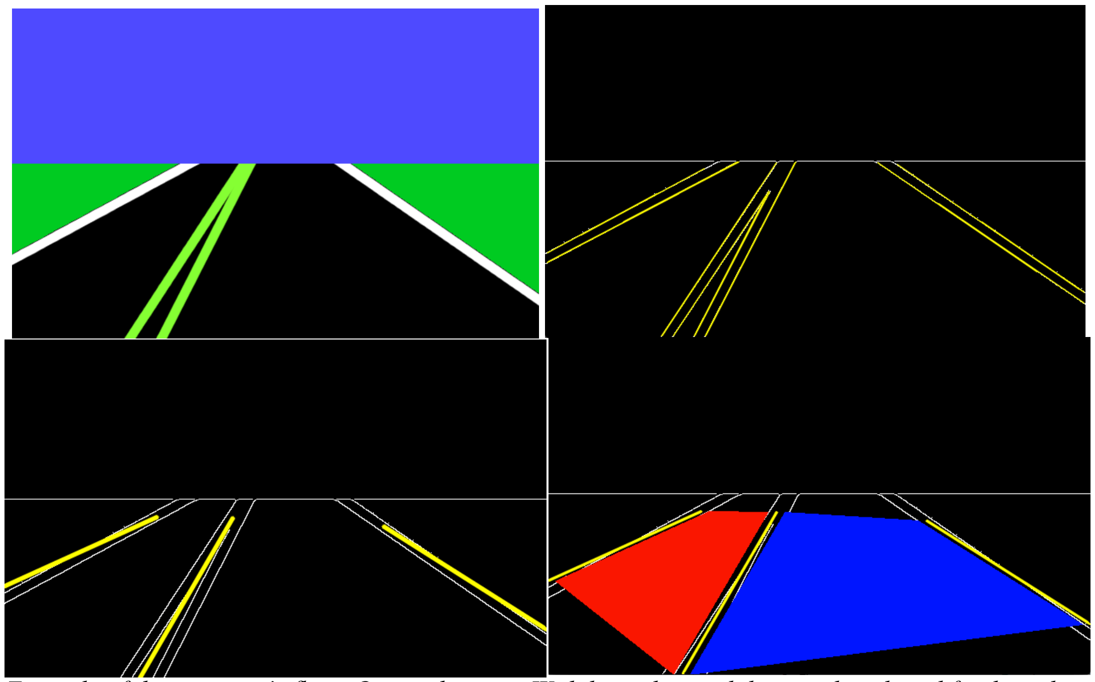
3a) *For adjacent lines: the nearest point on one line has a smaller magnitude than the farthest point on the closer line.*
3b) *For separated lines: the nearest point on one line has a larger magnitude than the farthest point on the closer line.*

A line is considered nearer if it’s y-coordinate is higher. This is because the origin (0,0) of an image is in the upper-left hand corner, as opposed to the common bottom-left. This is only true if their slopes are approximately the same, however. The farther point of a line is the point with a lesser y-magnitude.

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.

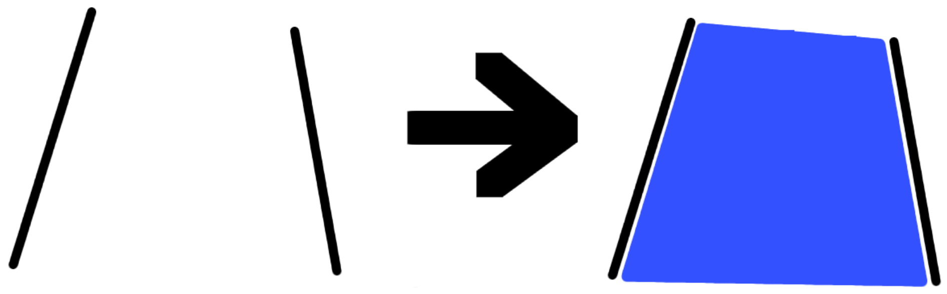

Example of the program’s flow: Original image, With lines detected, lines reduced, and final product

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`.

After reducing the image to only about three lines, the actual lanes must be drawn in. This is done by using draw_2lanes() and draw_1lane(). These functions call methods to determine which line is the middle, leftmost, and rightmost. Using this information, the OpenCV function fillPoly() is called to draw a filled polygon between the four points of these two lines, with a small gap between the line and actual start of the lane.



Example of drawing the lane

6) Implementation on the Pi

In order for this program to run on the Raspberry Pi, it must be compiled for it. This was accomplished by writing a Makefile, and putting the three main files in the folder with it: main.cpp, project.cpp, project.h.
Just like in the XCode implementation, the g++ compiler requires certain flags so that the OpenCV libraries can be added:

```
–lopencv_core  –lopencv_imgproc  –lopencv_highgui
–I /usr/local/include  –L /usr/local/lib
```

Another issue is how some OpenCV constants are handled in Linux versus Mac OS. The following changes were made in main.cpp:

```
CV_LOAD_IMAGE_GRAYSCALE => IMREAD_GRAYSCALE
CV_GRAY2RGB => COLOR_GRAY2RGB
CV_AA => LINE_AA
CV_IMWRITE_PNG_COMPRESSION => IMWRITE_PNG_COMPRESSION
```

The final major change in main.cpp was the filepath used for reading images. Since XCode has its own folder for handling additional files, only the filename needs to be specified, but since the Linux implementation does not, the ./images folder is used. So instead of just “road3.png”, “images/road3.png” is used when running the program with road3.png command-line argument. The “images/” tag is added in main.

The following is the Makefile used for this project. Running `make all` or `make install` will compile and install the project that can be run by performing `./opencv`.

```
# Makefile for Embedded Linux OpenCV project

# directory to store files in
DIRECTORY = ~/embedded_linux/project
# compiler flags (to link opencv libraries)
CFLAGS    =    -lopencv_core    -lopencv_imgproc    -lopencv_highgui    -I
/usr/local/include -L /usr/local/lib

all: install

install: project.o main.o
      mkdir -p $(DIRECTORY)
      g++ main.o project.o $(CFLAGS) -o opencv
      rm -rf *.o

project.o: project.cpp
      g++ -c project.cpp $(CFLAGS) -o project.o

main.o:     main.cpp
      g++ -c main.cpp $(CFLAGS) -o main.o

clean:
      rm -rf *.o opencv
```

Several shell scripts were written to handle importing the project folder to the Pi and retrieving the final output file (stored as images/output.png). These scripts perform the `scp` shell command to copy files between the Pi and my Mac. It included a parameter that would either `scp` to the local IP address or the external IP, depending on what network the script is run from.

7) Output

The program records the amount of time each step in the process takes, along with the total program runtime.

The output of the program, run on a Mac with OS 10.7.5, 2.53 GHz Intel Core 2 Duo and 8 GB of 1067MHz DDR3 RAM is as follows:

```
canny time: 0.016503 s
hough time: 0.020388 s
lines time: 4.1e-05 s
draw time:  0.022231 s
img time:   0.023424 s
TOTAL TIME: 0.088552 s
```

Canny time is how long the Canny edge detection method takes, Hough time is how long the HoughLinesP() function takes. Both are OpenCV functions.

Lines time is how long it takes for functions to reduce the number of lines from HoughLinesP to just two or three, and then extend them to the edge of the image. These are functions written by me.
Draw time is how long it takes for OpenCV to draw the lane lines and actual lane spaces, and Img time is how long it takes to generate the output image (a .png image). These are both OpenCV functions used within C++ functions I made.

APPENDIX

8) Header file

The following is the header file used for this project (`project.h`)

```
//
//  project.h
//  opencv
//
//  header file for Embedded Linux project

#ifndef opencv_project_h
#define opencv_project_h

//#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>

using namespace cv;
using namespace std;

// constants for opencv functions (canny, houghlinesp)
#define CANNY_T1        175
#define CANNY_T2        250
#define CANNY_APERTURE  3
#define HLINES_THRESH   40
#define HLINES_MINLINE  70
#define HLINES_MINGAP   30

// 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 horizontal
const double POINT_TOLERANCE = 100;          // how close (in px) for two lines “equal”
const double SLOPE_TOLERANCE = 0.70;         // how close (in %) two slopes “equal”
// tolerance for how close to edge to extend to a side of image (in px):
const int NEAR_EDGE = 100;



// ---
// for drawing lanes
// ---
// constants for drawing polygons:
const int LANE_EDGE = 10;                   // how far from lane lines should it be drawn
const int NUM_VERTICES = 4;                 // number of vertices for fillPoly()
const int NUM_POLYGONS = 1;                 // number of polygons to be drawn
const int LINE_TYPE = 8;                    // line type (8-point)
const double ALPHA = 0.10;                  // alpha, for blending in semi-transparent
const Scalar THISLANE_COLOR (255,0,0);      // color of lane photo taken in (blue)
const Scalar ONCOMING_COLOR (0,0,255);      // color of lane of oncoming traffic (red)
```

```
// draws the actual lanes in an image
Mat draw_2lanes(Mat, vector<Vec4i>);          // for more than 1 lane in the photo
Mat draw_1lane(Mat, vector<Vec4i>);           // for when there's only 1 lane in the photo
Vec4i middle_line(vector<Vec4i>);             // returns the "middle" line
Vec4i leftmost(vector<Vec4i>);                // returns the leftmost line
Vec4i rightmost(vector<Vec4i>);               // returns the rightmost line

// ---
// functions to reduce number of lines in image
// ---
void remove_horizontal(vector<Vec4i> *);      // removes horizontal lines from the vector
void remove_lines(int,int,vector<Vec4i>*);    // removes two lines from a vector of lines
bool horizontal(Vec4i);                       // determines if a line is horizontal
void remove_skylines(vector<Vec4i> *, int);   // removes lines if they are "in the sky"
bool skyline(Vec4i, int);                     // determines if a line is in the sky
// ---
vector<Vec4i> combine_lines(vector<Vec4i>); // combines adjacent/seperated lines
vector<Vec4i> extend_lines(vector<Vec4i>,int,int);  // extends lines to reach screen edge
// ---
bool greater_than(Vec4i, Vec4i);              // returns true if first line is higher up
void swap(Vec4i*, Vec4i*);                    // swaps two lines to pass to adj/sep()
// ---
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
// ---
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

#endif
```