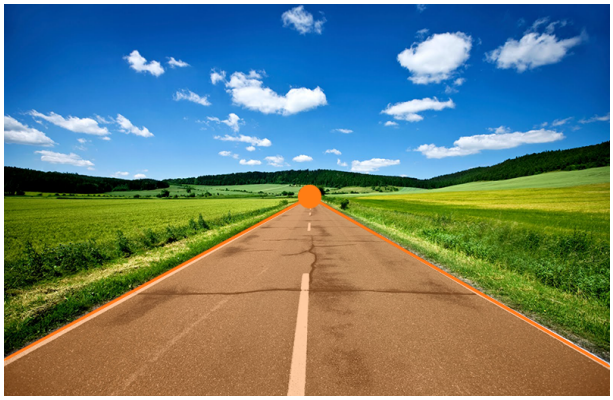
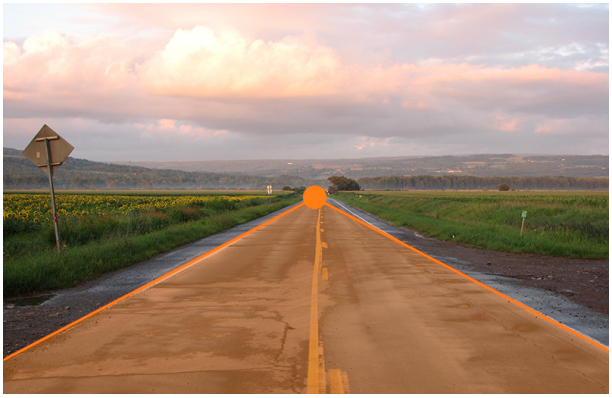
Annex

Results





Code Listing

Main.cpp

```
#include <iostream>
#include <opencv2/core.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include "DetectionTimer.h"
#include "RoadDetection.h"

using namespace std;
using namespace cv;

String baseAssetsPath = "../Assets/";

void displayHelp();
void processImage(String path);
void processVideo(String path);

int main(int argc, char** argv)
{
        if (argc < 3)
        {
                displayHelp();
                return -1;
        }

        int method = atoi(argv[1]);
        String file = (string)argv[2];

        switch (method)
        {
                case 1:
                        processImage(baseAssetsPath + file);
                        break;
                case 2:
                        processVideo(baseAssetsPath + file);
                        break;
                default:
                        displayHelp();
                        return -1;
        }

        waitKey(0);
}

void processImage(String path)
{
```

```
        RoadDetection roadDetector = RoadDetection(path);
        Mat result = roadDetector.processImage();

        namedWindow("Road Detection");
        imshow("Road Detection", result);
}

void processVideo(String path)
{
        VideoCapture cap = VideoCapture(path);
        RoadDetection roadDetector;

        namedWindow("Road Detection");

        if (!cap.isOpened())
        {
                cout << "Unable to read from file. Now exiting..." << endl;
                exit(1);
        }

        while (cap.isOpened() && waitKey(1) < 0)
        {
                Mat rawFrame;

                if (!cap.read(rawFrame))
                {
                        cout << "Skipped a frame..." << endl;
                }

                rawFrame = roadDetector.processVideo(rawFrame);

                imshow("Road Detection", rawFrame);
        }
}

void displayHelp()
{
        cout << "Usage example: RoadDetection.exe -method -filepath" << endl;
        cout << "Available methods: image(1), video(2)" << endl;
}
```

DetectionTimer.h

```
#pragma once
#include <opencv2/core.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

using namespace std;
using namespace cv;

class DetectionTimer
{
private:
        double initialTime;

public:
        DetectionTimer();
        ~DetectionTimer(){};

        void start();
        double getElapsed();
};
```

DetectionTimer.cpp

```
#include "DetectionTimer.h"

DetectionTimer::DetectionTimer()
{
        initialTime = 0;
}

void DetectionTimer::start()
{
        initialTime = (double)cvGetTickCount();
}

double DetectionTimer::getElapsed()
{
        double elapsed = (double)cvGetTickCount() - initialTime;
        return elapsed / ((double)cvGetTickFrequency() * 1000.);
}
```

Line.h

```
#pragma once
#include <iostream>
#include <opencv2/core.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>

using namespace std;
using namespace cv;

class Line
{
private:
        double getAngleBetweenPoints();
        float getSlope();
        float getIntercept();

public:
        Point pt1, pt2;
        double angle;
        float slope, intercept;

        Line(){};
        Line(Point pt1, Point pt2);
        ~Line(){};
};
```

Line.cpp

```
#include "Line.h"

Line::Line(Point pt1, Point pt2) :pt1(pt1), pt2(pt2)
{
        angle = getAngleBetweenPoints();
        slope = getSlope();
        intercept = getIntercept();
}

double Line::getAngleBetweenPoints()
{
        double xDiff = pt2.x - pt1.x;
        double yDiff = pt2.y - pt1.y;

        if (atan2(yDiff, xDiff) < 0)
        {
                return atan2(yDiff, xDiff) + 2 * CV_PI;
        }
        else
        {
                return atan2(yDiff, xDiff);
        }
}

float Line::getSlope()
{
        return (float)(pt2.y - pt1.y) / (pt2.x - pt1.x);
}

float Line::getIntercept()
{
        return pt1.y - slope * pt1.x;
}
```

RoadDetection.h

```
#pragma once
#include <iostream>
#include <opencv2/core.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include "opencv2/objdetect.hpp"
#include "DetectionTimer.h"
#include "Line.h"

using namespace std;
using namespace cv;

class RoadDetection
{
private:
        Mat original;
        Point previousVanishingPoint;
        vector<Line> previousLines;

        int BLUR_KERNEL = 5;
        int CANNY_MIN_THRESH = 50, CANNY_MAX_THRESH = 200;
        int HOUGH_DEFAULT_THRESH = 175, HOUGH_MIN_THRESH = 200, HOUGH_MIN_LINES
        int HOUGH_PROB_THRESH = 120, HOUGH_PROB_MIN_LINE_LENGTH = 80, HOUGH_PR
        int CASCADE_MIN_NEIGHBORS = 2, CASCADE_MIN_SIZE = 30, CASCADE_MAX_SIZE
        int LINE_DIST_MAX_DIFF = 20;

        float HOUGH_MIN_ANGLE = 1.35f, HOUGH_MAX_ANGLE = 1.75f;
        float HOUGH_PROB_MIN_ANGLE = 0.25f, HOUGH_PROB_MAX_ANGLE = 5.85f;
        double CASCADE_SCALE = 1.05f;
        double LINE_DIFF_MAX_ANGLE = 0.0872665;
        String CLASSIFIER_PATH = "../Assets/cars.xml";

        double getDistBetweenPoints(Point pt1, Point pt2);
        Point getPointAverage(Point pt1, Point pt2);
        Point getLineIntersection(Line l1, Line l2);
        Point getVanishingPoint(vector<Line> lines, Size frameSize);

        vector<Line> getHoughLines(Mat frame, bool useMinimum);
        vector<Line> getMainLines(vector<Line> lines);
        vector<Line> getHoughProbLines(Mat frame);
        vector<Line> getFilteredLines(vector<Line> lines);
        vector<Line> getLimitedLines(vector<Line> lines, int offset);
        vector<Line> shiftLines(vector<Line> lines, int shift);
        vector<Rect> getVehicles(Mat frame);
        vector<Point> getRoadShape(Mat screen, Line l1, Line l2, Point inter);

        void drawLines(Mat frame, vector<Line> lines, Scalar color, int thickne
```

```
        void drawCircle(Mat frame, Point center, Scalar color, int radius, int
        void drawRects(Mat frame, vector<Rect> rects, Scalar color, int thickne
        void drawRoadShape(Mat frame, vector<Point> points, Scalar color, doubl
        void combineWithSection(Mat frame, Mat section, int initialPos, int off

public:
        RoadDetection(){};
        RoadDetection(String filePath);
        RoadDetection(Mat original);
        ~RoadDetection(){};

        void setFile(String path);
        void setFile(Mat original);

        Mat processImage();
        Mat processVideo(Mat rawFrame);

        void displayControls();
};
```

RoadDetection.cpp

```
#include "RoadDetection.h"

/*****************************************************************
 * GENERAL CLASS METHODS
 *****************************************************************/

RoadDetection::RoadDetection(String filePath)
{
        original = imread(filePath);

        if (!original.data)
        {
                cout << "Unable to read from file. Now exiting..." << endl;
                exit(1);
        }
}

RoadDetection::RoadDetection(Mat frame)
{
        frame.copyTo(original);
}

void RoadDetection::setFile(String filePath)
{
        original = imread(filePath);

        if (!original.data)
        {
                cout << "Unable to read from file. Now exiting..." << endl;
                exit(1);
        }
}

void RoadDetection::setFile(Mat frame)
{
        frame.copyTo(original);
}

/*****************************************************************
* IMAGE/VIDEO PROCESSING
*****************************************************************/

/*
 * General image processing. Returns the original image with objects drawn on t
 */
Mat RoadDetection::processImage()
{
        Mat rawFrame, tmpFrame, grayFrame, blurredFrame, contoursFrame, houghFr
```

```
Point vanishingPoint;

vector<Line> houghLines, houghMainLines;
vector<Point> roadShape;

int sectionOffset;

// save a copy of the original frame
original.copyTo(rawFrame);

// smooth and remove noise
GaussianBlur(rawFrame, blurredFrame, Size(BLUR_KERNEL, BLUR_KERNEL), 0,

// edge detection (canny, inverted)
Canny(blurredFrame, contoursFrame, CANNY_MIN_THRESH, CANNY_MAX_THRESH);
threshold(contoursFrame, contoursFrame, 128, 255, THRESH_BINARY);

// hough transform lines
houghLines = getHoughLines(contoursFrame, true);

// vanishing point
vanishingPoint = getVanishingPoint(houghLines, rawFrame.size());
sectionOffset = vanishingPoint.y;

// section frame (below vanishing point)
contoursFrame.copyTo(tmpFrame);
sectionFrame = tmpFrame(CvRect(0, vanishingPoint.y, contoursFrame.cols,

// re-apply hough transform to section frame
houghLines = getHoughLines(sectionFrame, true);

// shift lines downwards
houghLines = shiftLines(houghLines, sectionOffset);

houghLines = getFilteredLines(houghLines);

// best line matches
houghMainLines = getMainLines(houghLines);

if (houghMainLines.size() >= 2)
{
        Point intersection = getLineIntersection(houghMainLines[0], hou

        if (intersection.x > 0 && intersection.y >= 0)
        {
                vanishingPoint = intersection;
                sectionOffset = intersection.y;
        }

        // get road shape
```

```
                roadShape = getRoadShape(rawFrame, houghMainLines[0], houghMain
        }

        // limit lines
        houghLines = getLimitedLines(houghLines, sectionOffset);
        houghMainLines = getLimitedLines(houghMainLines, sectionOffset);

        // drawing process
        drawLines(rawFrame, houghLines, Scalar(0, 0, 255), 2, 0);
        drawLines(rawFrame, houghMainLines, Scalar(20, 125, 255), 2, 0);
        drawRoadShape(rawFrame, roadShape, Scalar(20, 125, 255), 0.4);
        drawCircle(rawFrame, vanishingPoint, Scalar(20, 125, 255), 15, -1, 0);

        return rawFrame;
}

/*
 * General videos processing. @TODO Make use of previous frames?
 */
Mat RoadDetection::processVideo(Mat rawFrame)
{
        Mat originalFrame, tmpFrame, grayFrame, blurredFrame, contoursFrame, ho
        Point vanishingPoint;

        vector<Line> houghLines, houghMainLines;
        vector<Point> roadShape;
        vector<Rect> vehicles;

        int sectionOffset;

        // convert to grayscale
        cvtColor(rawFrame, grayFrame, CV_BGR2GRAY);
        equalizeHist(grayFrame, grayFrame);

        // smooth and remove noise
        GaussianBlur(grayFrame, blurredFrame, Size(BLUR_KERNEL, BLUR_KERNEL), 0

        // edge detection (canny, inverted)
        Canny(blurredFrame, contoursFrame, CANNY_MIN_THRESH, CANNY_MAX_THRESH);
        threshold(contoursFrame, contoursFrame, 128, 255, THRESH_BINARY);

        // hough transform
        houghLines = getHoughLines(contoursFrame, true);

        // vanishing point
        vanishingPoint = getVanishingPoint(houghLines, rawFrame.size());
        sectionOffset = vanishingPoint.y;

        // if we can't find a point, use the one from a previous frame
        if (vanishingPoint.x == 0 && vanishingPoint.y == 0)
```

```
{
        vanishingPoint = previousVanishingPoint;
}
// if we can, save it for future use
else
{
        previousVanishingPoint = vanishingPoint;
}

// section frame (below vanishing point)
sectionFrame = contoursFrame(CvRect(0, sectionOffset, contoursFrame.col

// re-apply hough transform to section frame
houghLines = getHoughLines(sectionFrame, true);

// shift lines downwards
houghLines = shiftLines(houghLines, sectionOffset);

// best line matches
houghMainLines = getMainLines(houghLines);

// update vanishing point according to the new section
if (houghMainLines.size() >= 2)
{
        previousLines = houghMainLines;
}
else
{
        houghMainLines = previousLines;
}

if (houghMainLines.size() >= 2)
{
        Point intersection = getLineIntersection(houghMainLines[0], hou

        if (intersection.x > 0 && intersection.x < rawFrame.cols && int
        {
                vanishingPoint = intersection;
                sectionOffset = intersection.y;
        }

        // get road shape
        roadShape = getRoadShape(rawFrame, houghMainLines[0], houghMain
}

// limit lines
houghLines = getLimitedLines(houghLines, sectionOffset);
houghMainLines = getLimitedLines(houghMainLines, sectionOffset);

// drawing frame and vehicles
```

```
        drawingFrame = rawFrame(CvRect(0, sectionOffset, contoursFrame.cols, co
        // vehicles = getVehicles(drawingFrame);

        // drawing process
        drawLines(rawFrame, houghLines, Scalar(0, 0, 255), 2, 0);
        drawLines(rawFrame, houghMainLines, Scalar(20, 125, 255), 2, 0);
        drawRoadShape(rawFrame, roadShape, Scalar(20, 125, 255), 0.3);
        drawCircle(rawFrame, vanishingPoint, Scalar(20, 125, 255), 15, -1, 0);
        // drawRects(rawFrame, vehicles, Scalar(255, 0, 0), 1, sectionOffset);

        return rawFrame;
}

/*****************************************************************
* DRAWING
*****************************************************************/

void RoadDetection::drawLines(Mat frame, vector<Line> lines, Scalar color, int
{
        for (size_t i = 0; i < lines.size(); i++)
        {
                Point pt1 = lines[i].pt1;
                Point pt2 = lines[i].pt2;

                pt1.y += offset;
                pt2.y += offset;

                line(frame, pt1, pt2, color, thickness, CV_AA);
        }
}

void RoadDetection::drawCircle(Mat frame, Point center, Scalar color, int radiu
{
        center.y += offset;

        if (center.x > 0 && center.y > 0)
        {
                circle(frame, center, radius, color, thickness, CV_AA);
        }
}

void RoadDetection::drawRects(Mat frame, vector<Rect> rects, Scalar color, int
{
        for (size_t i = 0; i < rects.size(); i++)
        {
                rects[i].y += offset;
                rectangle(frame, rects[i], color, thickness, CV_AA);
        }
}
```

```
void RoadDetection::drawRoadShape(Mat frame, vector<Point> points, Scalar color
{
        Mat copy;
        Point shapePoints[1][3];
        int npt[] = { 3 };

        if (points.size() <= 0)
        {
                return;
        }

        for (size_t i = 0; i < points.size(); i++)
        {
                shapePoints[0][i] = points[i];
        }

        const Point* ppt[1] = { shapePoints[0] };
        frame.copyTo(copy);

        fillPoly(copy, ppt, npt, 1, color, CV_AA);
        addWeighted(copy, alpha, frame, 1.0 - alpha, 0.0, frame);
}

void RoadDetection::combineWithSection(Mat frame, Mat section, int initialPos,
{
        for (int i = initialPos; i < section.rows; i++)
        {
                for (int j = 0; j < section.cols; j++)
                {
                        Vec3b pixel = section.at<Vec3b>(i, j);
                        frame.at<Vec3b>(i + offset, j) = pixel;
                }
        }
}

/*****************************************************************
* POINT/LINE AUXILIARS
*****************************************************************/

double RoadDetection::getDistBetweenPoints(Point pt1, Point pt2)
{
        return sqrt(pow(pt2.x - pt1.x, 2) + pow(pt2.y - pt1.y, 2));
}

/*
 * Returns a new points where x and y are the respective averages of the values
 */
Point RoadDetection::getPointAverage(Point pt1, Point pt2)
{
        return Point((pt1.x + pt2.x)/2, (pt1.y + pt2.y)/2);
```

```
}

/*
 * Returns the points where two lines intersect. If the lines have the same slo
 * Makes use of y = mx + c, where m = slope, c = intercept.
 */
Point RoadDetection::getLineIntersection(Line l1, Line l2)
{
        Point l1p1 = l1.pt1;
        Point l1p2 = l1.pt2;
        Point l2p1 = l2.pt1;
        Point l2p2 = l2.pt2;

        float l1Slope = l1.slope;
        float l2Slope = l2.slope;

        if (l1Slope == l2Slope)
        {
                return Point(0, 0);
        }

        float l1Inter = l1.intercept;
        float l2Inter = l2.intercept;

        float interX = (l2Inter - l1Inter) / (l1Slope - l2Slope);
        float interY = (l1Slope * interX + l1Inter);

        Point interPoint = Point((int)interX, (int)interY);

        return interPoint;
}

vector<Line> RoadDetection::getLimitedLines(vector<Line> lines, int offset)
{
        vector<Line> output;

        for (size_t i = 0; i < lines.size(); i++)
        {
                Line line = lines[i];
                int newX = (int)((offset - line.intercept) / line.slope);

                if (line.slope < 0)// /.
                {
                        line.pt2.x = newX;
                        line.pt2.y = offset;
                }
                else // \.
                {
                        line.pt1.x = newX;
                        line.pt1.y = offset;
```

```
                }

                output.push_back(line);
        }

        return output;
}

/*****************************************************************
* POINT/LINE AUXILIARS
*****************************************************************/

/*
 * Returns a vector of the lines obtained from the Hough Transform.
 * @param useMinimum if enabled, the voting threshold is lowered until HOUGH_M
 */
vector<Line> RoadDetection::getHoughLines(Mat frame, bool useMinimum)
{
        vector<Vec2f> lines;
        vector<Line> filteredLines;

        // if using  minimum, repeat hough process while lowering the voting u
        if (useMinimum)
        {
                int thresh = HOUGH_MIN_THRESH;

                while ((int)lines.size() < HOUGH_MIN_LINES && thresh > 0)
                {
                        HoughLines(frame, lines, 1, CV_PI / 180, thresh, 0, 0);
                        thresh -= 5;
                }
        }

        // otherwise, run hough process once using a default voting value
        else
        {
                HoughLines(frame, lines, 1, CV_PI / 180, HOUGH_DEFAULT_THRESH,
        }

        // convert rho and theta to actual points
        for (size_t i = 0; i < lines.size(); i++)
        {
                float rho = lines[i][0];
                float theta = lines[i][1];

                if (theta < HOUGH_MIN_ANGLE || theta > HOUGH_MAX_ANGLE)
                {
                        Point pt1, pt2;
                        double a = cos(theta);
                        double b = sin(theta);
```

```
                        double x0 = a*rho;
                        double y0 = b*rho;

                        pt1.x = cvRound(x0 + 1000 * (-b));
                        pt1.y = cvRound(y0 + 1000 * (a));
                        pt2.x = cvRound(x0 - 1000 * (-b));
                        pt2.y = cvRound(y0 - 1000 * (a));

                        Line line = Line(pt1, pt2);
                        filteredLines.push_back(line);
                }
        }

        return filteredLines;
}

/*
 * Returns a vector of the lines obtained from the Hough Probabilistic Transfor
 * Horizontal lines are automatically removed and not returned (values can be a
 */
vector<Line> RoadDetection::getHoughProbLines(Mat frame)
{
        vector<Vec4i> lines;
        vector<Line> filteredLines;

        HoughLinesP(frame, lines, 1, CV_PI / 180, HOUGH_PROB_THRESH, HOUGH_PRO

        for (size_t i = 0; i < lines.size(); i++)
        {
                Point pt1 = Point(lines[i][0], lines[i][1]);
                Point pt2 = Point(lines[i][2], lines[i][3]);
                Line line = Line(pt1, pt2);

                if (line.angle > HOUGH_PROB_MIN_ANGLE && line.angle < HOUGH_PRO
                {
                        filteredLines.push_back(line);
                }
        }

        return filteredLines;
}

/*
 * Returns the vanishing point for a group of lines based on the intersections
 * Lines with a higher angle difference in between them have a higher weight in
 */
Point RoadDetection::getVanishingPoint(vector<Line> lines, Size frameSize)
{
        vector<Point> interPoints;
        vector<double> interAngles;
```

```
double interAnglesSum = 0;
Point vanishPoint = Point(0, 0);

if (lines.size() <= 0)
{
        return vanishPoint;
}

for (size_t i = 0; i < lines.size() - 1; i++)
{
        for (size_t j = i + 1; j < lines.size(); j++)
        {
                Line l1 = lines[i];
                Line l2 = lines[j];

                Point l1p1 = l1.pt1;
                Point l1p2 = l1.pt2;
                Point l2p1 = l2.pt1;
                Point l2p2 = l2.pt2;

                Point interPoint = getLineIntersection(l1, l2);

                if (interPoint.x > frameSize.width || interPoint.x <= 0
                {
                        continue;
                }

                double interAngle = abs(l1.angle - l2.angle);
                interAnglesSum += interAngle;

                interPoints.push_back(interPoint);
                interAngles.push_back(interAngle);
        }
}

if (interPoints.size() <= 0)
{
        return vanishPoint;
}

for (size_t i = 0; i < interPoints.size(); i++)
{
        Point interPoint = interPoints[i];
        double interAngle = interAngles[i];
        double weight = interAngle / interAnglesSum;

        vanishPoint.x += cvRound(interPoint.x * weight);
        vanishPoint.y += cvRound(interPoint.y * weight);
}
```

```
        return vanishPoint;
}

/*
 * Combines all pairs of lines with similar angles and low distance in between
 * This process is repeated recursively until no more lines can be combined.
 */
vector<Line> RoadDetection::getFilteredLines(vector<Line> lines)
{
        vector<Line> output;
        vector<bool> linesToProcess(lines.size(), true);

        for (size_t i = 0; i < lines.size() - 1; i++)
        {
                if (!linesToProcess[i])
                {
                        continue;
                }

                for (size_t j = i + 1; j < lines.size(); j++)
                {
                        Line l1 = lines[i];
                        Line l2 = lines[j];

                        if (!linesToProcess[j])
                        {
                                continue;
                        }

                        double angDiff = abs(l1.angle - l2.angle);

                        if (angDiff < LINE_DIFF_MAX_ANGLE)
                        {
                                double distpt1 = getDistBetweenPoints(l1.pt1, l
                                double distpt2 = getDistBetweenPoints(l1.pt2, l

                                if (distpt1 < LINE_DIST_MAX_DIFF || distpt2 < L
                                {
                                        Line combinedLine = Line(getPointAverag

                                        output.push_back(combinedLine);

                                        linesToProcess[i] = false;
                                        linesToProcess[j] = false;

                                        break;
                                }
                        }
```

```
                        if (j == (lines.size() - 1))
                        {
                                output.push_back(l1);
                        }
                }
        }

        int lastIndex = lines.size() - 1;

        if (linesToProcess[lastIndex])
        {
                output.push_back(lines[lastIndex]);
        }

        if (output.size() < lines.size())
        {
                return getFilteredLines(output);
        }

        return output;
}

/*
 * Returns the two lines with the highest angle difference in between them.
 * Generally, these are the external lines that define a road.
 * @TODO aren't these the lines with the highest and lowest slope?
 */
vector<Line> RoadDetection::getMainLines(vector<Line> lines)
{
        vector<Line> mainLines;
        Line mainLine1, mainLine2;
        double bestAngle = -1.f;

        if ((int)lines.size() < 2)
        {
                return mainLines;
        }

        for (size_t i = 0; i < lines.size() - 1; i++)
        {
                for (size_t j = i + 1; j < lines.size(); j++)
                {
                        Line l1 = lines[i];
                        Line l2 = lines[j];

                        Point l1p1 = l1.pt1;
                        Point l1p2 = l1.pt2;
                        Point l2p1 = l2.pt1;
                        Point l2p2 = l2.pt2;
```

```
                        double angleDiff = abs(l1.angle - l2.angle);

                        if (angleDiff > bestAngle)
                        {
                                bestAngle = angleDiff;
                                mainLine1 = l1;
                                mainLine2 = l2;
                        }
                }
        }

        mainLines.push_back(mainLine1);
        mainLines.push_back(mainLine2);

        return mainLines;
}

/*
 *
 */
vector<Point> RoadDetection::getRoadShape(Mat screen, Line l1, Line l2, Point i
{
        vector<Point> output;

        int screenSizeX = screen.cols;
        int screenSizeY = screen.rows;

        //simple side detection
        Line leftLine;
        Line rightLine;

        if (l1.slope > 0)
        {
                leftLine = l1;
                rightLine = l2;
        }
        else
        {
                leftLine = l2;
                rightLine = l1;
        }

        //Start filling point vector

        output.push_back(inter);

        // y = mx + b <=> screenSizeY = mx + b <=> x = (screenSizeY - b) / m
        Point leftPoint = Point((int)((screenSizeY - leftLine.intercept) / left
        output.push_back(leftPoint);
```

```
        Point rightPoint = Point((int)((screenSizeY - rightLine.intercept) / ri
        output.push_back(rightPoint);

        return output;
}

/*
 * Returns a list of all the vehicles found in a frame.
 * The base classifier used for this process can be set in CLASSIFIER_PATH.
 */
vector<Rect> RoadDetection::getVehicles(Mat frame)
{
        Mat copy, equalizedFrame;
        CascadeClassifier vehiclesCascade;
        vector<Rect> vehicles;

        frame.copyTo(copy);

        cvtColor(copy, equalizedFrame, CV_BGR2GRAY);
        equalizeHist(equalizedFrame, equalizedFrame);

        vehiclesCascade.load(CLASSIFIER_PATH);
        vehiclesCascade.detectMultiScale(equalizedFrame, vehicles, CASCADE_SCAL

        return vehicles;
}

vector<Line> RoadDetection::shiftLines(vector<Line> lines, int shift)
{
        for (size_t i = 0; i < lines.size(); i++)
        {
                lines[i].pt1.y += shift;
                lines[i].pt2.y += shift;
                lines[i].intercept += shift;
        }

        return lines;
}
```