Rheinisch-Westfälische Technische Hochschule Aachen
Computer science chair 3

Autonomous driving lab documentation

Vanishing Point Detection

Mike Lorang

Uma Moothiringote

Jiahui Geng

Christoph Richter

Supervisor: Evgeny Kusmenko

Contents

1 Vanishing Point Detection **4**

1.1 Motivation . . . 4

1.2 Preprocessing the image . . . 4

1.2.1 Detecting the ROI with texture detection . . . 4

1.2.2 Detecting lines in the preprocessed image . . . 6

1.2.3 Calculation of the vanishing angle . . . 7

1.2.4 Applying a filter to the detected vanishing angles . . . 7

1.3 Lane Tracking . . . 7

1.3.1 ROI . . . 8

1.3.2 Lane Detection . . . 8

1.3.3 Lane Tracking . . . 9

2 Object Detection And Tracking **10**

2.1 Architecture . . . 10

2.2 Car Detection . . . 11

2.3 Pedestrian Detection . . . 13

2.4 Classifier Training . . . 15

2.4.1 Get Training Data . . . 15

2.4.2 Train Cascade Classifier . . . 16

2.5 Depth estimation . . . 17

2.5.1 Stereo Camera . . . 17

2.5.2 Stereo Reconstruction . . . 17

2.6 Tracking . . . 18

2.7 Possible Optimizations . . . 20

2.8 Remarks . . . 20

3 Integration **20**

3.1 Integration of OpenCV . . . 20

3.2 Integration of computer vison components . . . 20

List of Tables

List of Figures

1 Original image . . . 5
2 Mask created . . . 6
3 Blue horizontal lines detected for the road area in the image . . . 6
4 ROI extraction with manual mask . . . 8
5 Houghlines from HoughlinsP . . . 9
6 Lane detection with considerable results . . . 9
7 Class diagram architecture . . . 10
8 Class diagram DetectedObject . . . 11
9 HAAR features. . . . 12
10 LBP features. . . . 12
11 Boost training. . . . 13
12 HOG features. . . . 14
13 SVM training. . . . 15
14 opencv_traincascade . . . 17
15 Disparity to depth. . . . 18
16 Nearest neighbour gating . . . 19
17 Class diagram tracking . . . 19

1 Vanishing Point Detection

1.1 Motivation

The overall goal of the vanishing point detection is to help the autonomous car to find the right steering angle. The Main tasks were the detection of the lane, filtering bad results and detecting the vanishing angle with the given data. These goals are achieved doing the following steps:

- Preprocessing the image
- Detecting the ROI with texture detection
- Detecting lines in the preprocessed image
- Calculation of the vanishing angle
- Applying a filter to the detected vanishing angles.

1.2 Preprocessing the image

The preprocessing is done in the methods *Mat edgeDetect(Mat)* and in *Mat getROI(Mat)*. Both methods are in the LaneDetection Object.

The method *Mat edgeDetect(Mat)* takes the original image, converts is to grayscale and applies an Canny edge detection with the parameters *cannyHigh* and *cannyLow*.

Moreover will the image we split vertically in the middle to assure that one lane-mark on the left and one lane-mark on the right is detected. This happens in the *houghlines* and in the *houghlinesP*.
Either one of these two methods is used for the detection of the lane-marks.

1.2.1 Detecting the ROI with texture detection

ROI is generated using the method detectEdge() in EdgeDetector object. The motivation behind using texture detection to decide on the ROI for lane detection is that roads are generally grey in colour. The colour *grey* generally has almost the same values for R,G, B and this information is made use of in this algorithm. A gradient is taken between the RGB values to determine whether the area is grey or non-grey: a near-zero gradient implies a grey area.
In the implementation, we have utilised the saturation from the HSV model for the grey detection; grey areas have generally near-zero saturation. On the saturation of the image, a

Sobel derivative is taken in the x and y directions (sobel_x and sobel_y). Pseudo-equations are given below.

$$sobel_x = Sobel(s, direction = x) \tag{1}$$
$$sobel_y = Sobel(s, direction = y) \tag{2}$$
where s is the saturation from HSV model of the image

Texture anisotropy is carried out on the image using the method described here: A sliding window of size 3 (variable *WINDOW_SIZE* of class EdgeDetector) is passed through $sobel_x$ and $sobel_y$ to give win_x and win_y and a variance-covariance matrix is formed for each sliding window:

$$\Sigma = \begin{bmatrix} \sigma_{win_x win_x} & \sigma_{win_x win_y} \\ \sigma_{win_y win_x} & \sigma_{win_y win_y} \end{bmatrix} \tag{3}$$
where σ_{ab} is the covariance between a and b (4)

When a region is grey, the eigen values (λ_1 and λ_2) of the variance-covariance matrix will both be smaller values. This is returned by the function varCovar(Mat winX, Mat winY) where winX and winY are the current positions of the sliding windows on $sobel_x$ and $sobel_y$. For each of the sliding windows, we calculate the anisotropy by calculating the difference in eigen values of the variance-covariance matrix:

$$anisotropy = \frac{\lambda_1^2 - \lambda_2^2}{\lambda_1} \tag{5}$$

Anisotropy can range from 0 to 1, 1 being the strongest. We create a mask from the anisotropy values such that we all values above the average anisotropy is 1 and the rest 0. We expect the road region to have weaker anisotropy. The mask is eroded and dilated to remove noise. Figure 1 shows the original picture and Figure 2 shows the output of the mask after the algorithm is applied and anisotropy is calculated.

Figure 1: Original image

Figure 2: Mask created

The next task is to find the road area from the mask. For this, we start from the bottom of the mask to find continuous regions that have 0 value in the mask (the road is expected to be empty right in front of the vehicle). Once we get the 2 end points on the left and right sides of the image for continuous 0 values, we start searching from there above to find such end points in each row of pixels, so that by the end of the search, we have a set of line segments that cover the road area. This is done is drawHorizLines2() in EdgeDetector. A sample result of the algorithm is included below.

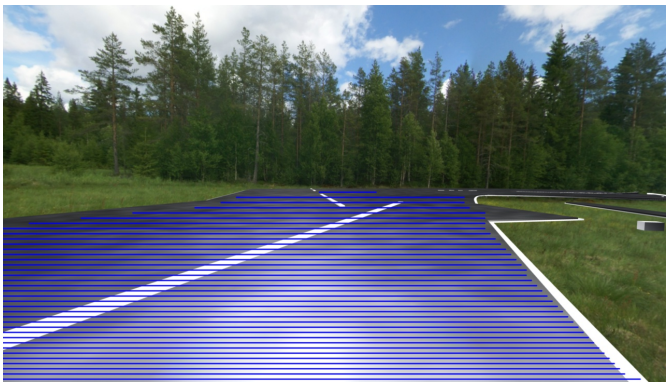

Figure 3: Blue horizontal lines detected for the road area in the image

The set of points are converted into a bounding box which is considered as the ROI for the lane detection explained in the later sections. If the method fails to return a bounding box, the following algorithm uses the full image for further processing.

1.2.2 Detecting lines in the preprocessed image

There two slightly different methods which are implemented for detecting the lines and either one can be used. One method is the houghlines transformation and the other one is the probabilistic houghlines transformation. In my experience the probabilistic houghlines perform a bit better but I think that the houghlinesP are also a bit more sensible to noise

since it is able detect smaller lines.
The method *double houghlines(Mat,Mat)* and *double houghlinesP(Mat,Mat)* work exactly the same except for the function used from the OpenCV library. As mentioned in 1.2 the image is split in two parts. Then the houghlines line-detection function is called to detect the lines on each halve of the preprocessed image separately. To prevent noise the variable *maxiterations* limits the number of lines which can be detected on each side to three. Then there is a loop which converts the lines from the Hough space to the Cartesian space and checks if the gradient of the detected line fits is in between two predefined values (gradientHigh and gradientLow). These two can be adjusted to filter some more noise. I was not able to test the algorithm on many different datasets that is why these parameters might need to be adjusted.
Finally for the left and the right halves the lines fitting the criteria are added to a list of lines(*leftLinesList* and *rightLinesList*). The lines objects has two parameters m and c where m is the gradient of the line and c is the distance of the intersection of the y-axis and the origin. These two lists are used to calculate the vanishing angle.

1.2.3 Calculation of the vanishing angle

The method *getVanishingAngleOfPreprocessedImg* takes the lists of lines on the right and the left of the right plus the original image and calculates the vanishing angle. If one of the lists are empty it simply returns zero. The average intersection point is calculated in a for loop. The double x is the x coordinate of the intersection and y is he y coordinate of the intersection. These are summed up for every intersection and divided by the number of intersections to get the average intersection point.
Having that point it is possible to calculate the vanishing angle. The variable *dx* is the difference of the middle of the picture and the x coordinate of the vanishing point and *dy* is the difference between the image height and the y coordinate of the vanishing point (since the origin of the coordinate system is on the top left of the picture). Moreover the hypotenuse is calculated. Now the vanishing angle is calculated and returned with help of these parameters.

1.2.4 Applying a filter to the detected vanishing angles

The method *getVanishingAngle* puts all the methods together and calculates the filtered vanishing angle of a given image. When it is called more than once in the program is filters the vanishing angle. The method keeps track of the last 5 vanishing angles and weights them with the Fibonacci numbers.

1.3 Lane Tracking

All the algorithm steps are integrated in function getLaneVaule(), the input are the Mat from the origin image and the output are the lane information, a array with 8 double numbers. The first 4 numbers are right line information and the last 4 are for left line.lane[0-3] is a vector of 4 elements (like Vec4f) - (vx, vy, x0, y0), where (vx, vy) is a normalized vector co-linear to the line and (x0, y0) is a point on the line. According to these information we can definitely define the lane region and the direction information in order to support the autonomous driving.

1.3.1 ROI

If we directly use edge detectors, we will find the shadow, the intersection between the forest and the ground will be considered as edges of lanes, however especially for lane detection, the only part we are interested is only the trapezium. This part corresponds to the rectangle area of lanes but with some distortion because of perspective view.
One ROI method was referred above, another alternative method to define is to use the mask that we define ourselves. As from the samples we got from visualization group, the camera scans the lane always with a specific angle, therefore we choose such a polygon area which contains enough information for lane detection as ROI.

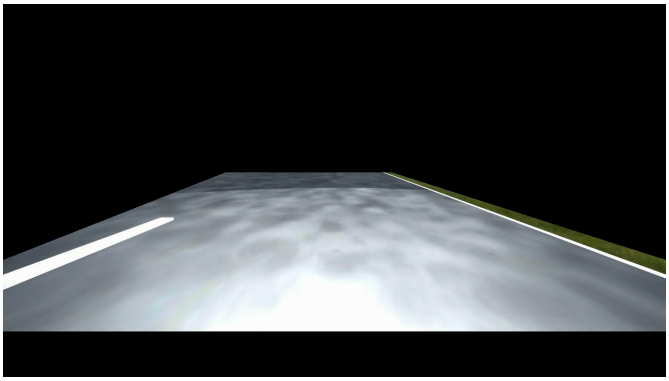

Figure 4: ROI extraction with manual mask

1.3.2 Lane Detection

The core of lane detection is to use HoughlinesP to find the lane, though due to the distortion of image and the lane curve in the real life, the extracted lines from the lane lines cannot be strictly straight, however if we just pick image segment which is enough near from the camera, they can be considered as straight lines. The reason we choose HoughlinesP is because it is faster and more effective because its probabilistic method. The lane detection contains first the canny edge detection. We first convert the original image into gray image and implement the canny detector to find the edge.Then we perform the HoughlineP on this image and the result is as followed:

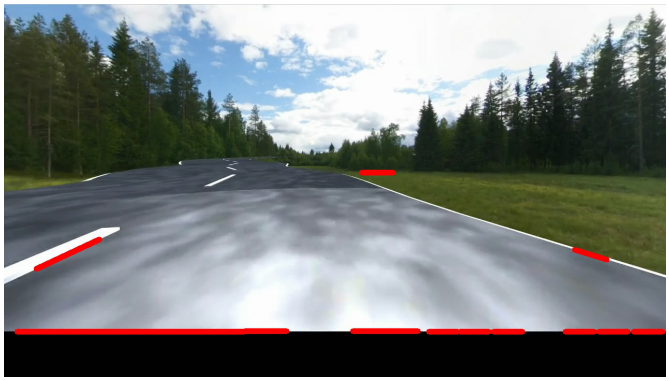

Figure 5: Houghlines from HoughlinsP

When we check this picture, the two lines for lane edges are drawn, however some other edges like the edges caused by the shadow and the lines below are also computed. Then we can use the slope of the lines as the criterion to eliminate the lines that are not the edges.
Because the light and lane condition, in some cases one long lane line can be detected as several different short lines, the method we handle it is to extract the endpoint of those lines and use fitLine from openCV to fit the points and finally get the edges which define the region of lane.

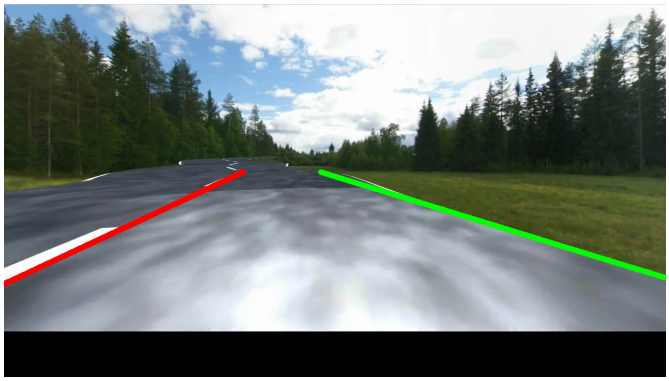

Figure 6: Lane detection with considerable results

1.3.3 Lane Tracking

In the interval of two lane lines, as above, the left line will not be detected. We use the latest detected line instead. So along the lane every frame will have exact two lines.The effect of this algorithm are showed url: https://www.youtube.com/watch?v=wtGzSp859xQ

2 Object Detection And Tracking

2.1 Architecture

Class diagram 7 describes the architecture of the object detection classes and their integration in the framework.

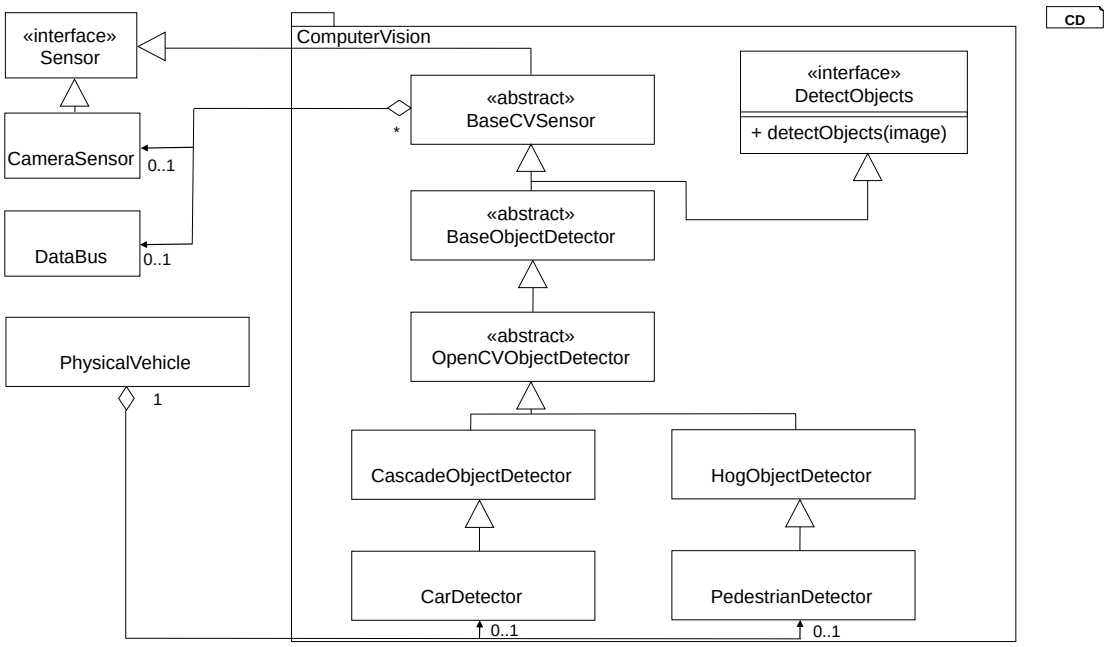

Figure 7: Class diagram architecture

BaseCVSensor Basis class for all computer vision component. Implements the sensor interface and provides a CameraSensor.

BaseObjectDetector Basis class for all object detectors. Implements DetectObject. Writes results on the bus. Executes object detection.

nCVObjectDetector Basis class for object detectors based on OpenCV. Initialises the OpenCV context. Provides a GUI. Implements angle and depth detection to object.

cadeObjectDetector Performs OpenCV cascade object detection according to the loaded cascade object classifier. Classifiers must be cascade classifier supporting HAAR features or LBP (local binary pattern) features from OpenCV.

HOGObjectDetector Performs OpenCV object detection using HOG (Histogram of Oriented Gradients) features. Classifier is a HOG classifier.

CarDetector Detects cars. Loads a trained cascade classifier.

CarDetector Detects pedestrians. Uses OpenCVs default pedestrian classifier or a classifier trained by Daimler based on HOGs.

The method detectObjects(image) returns a list of "DetectedObject"s. Detected objects provides the following information shown in class diagram 8.

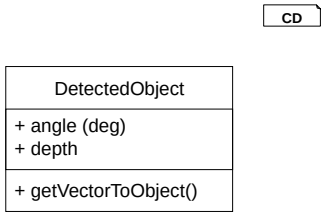

Figure 8: Class diagram DetectedObject

DetectedObject Represents a detected object by angle to the object and the estimated depth (distance to the object). Is used to further process the object detection results.

2.2 Car Detection

Car detection is implemented by using OpenCV 3.2. OpenCV provides a object detection function which uses Cascade Classifier. This object detection function is based on the sliding window approach and is a car/non car classification (binary). The classifers are trained by Boosting approaches (e.g AdaBoost).

Used features are HAAR features or optional LBP (local binary patterns)

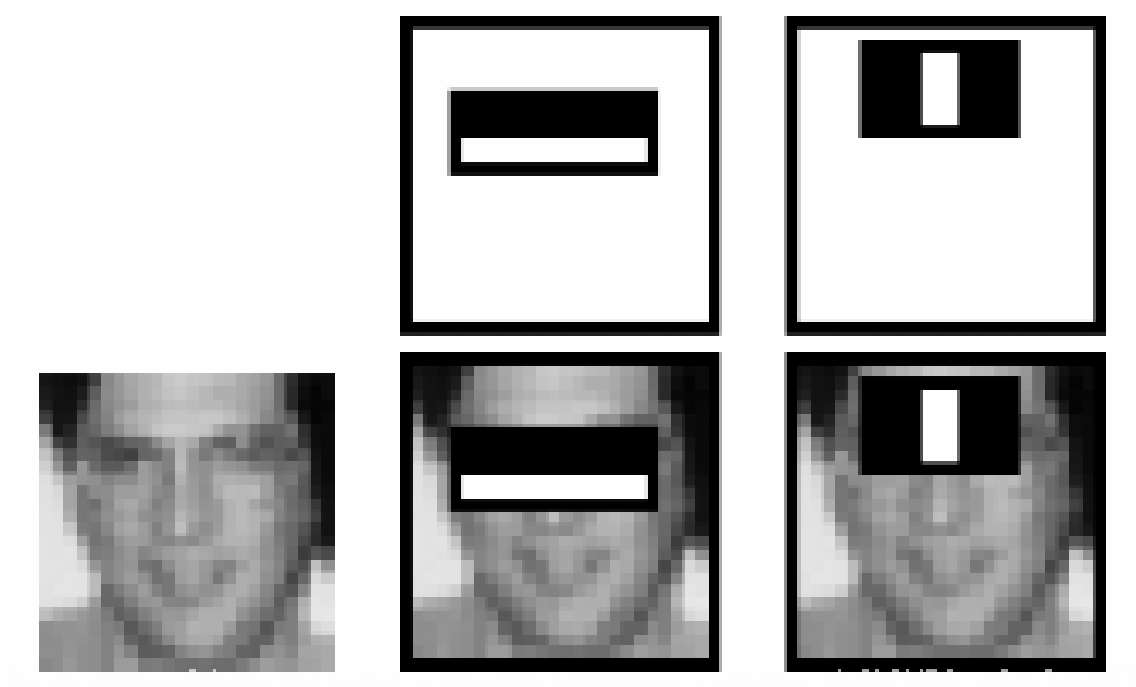

Figure 9: HAAR features.
Source: http://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html

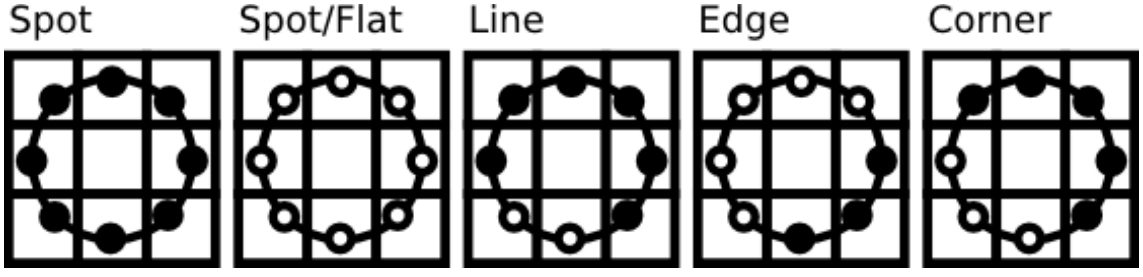

Figure 10: LBP features.
Source: http://docs.opencv.org/2.4/modules/contrib/doc/facerec/facerec_tutorial.html

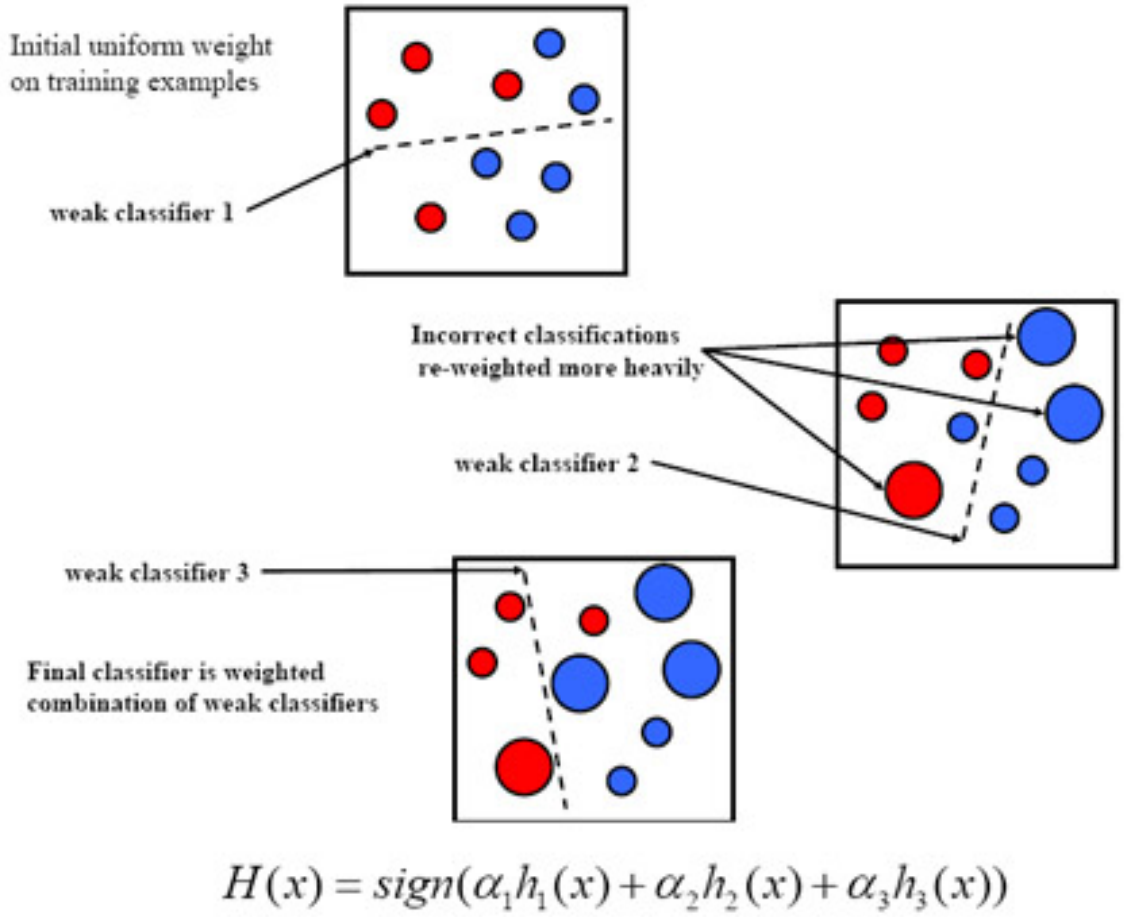

Figure 11: Boost training.
Source: http://www.cc.gatech.edu/ kihwan23/imageCV/Final2005/FinalProject_KH.htm

2.3 Pedestrian Detection

Pedestrian detection also uses a object detection function by OpenCV 3.2. Instead of HAAR features HOGs (Histograms of Oriented Gradients) are used as features. These are usually trained using SVMs. OpenCV provides such trained classifier for pedestrians.

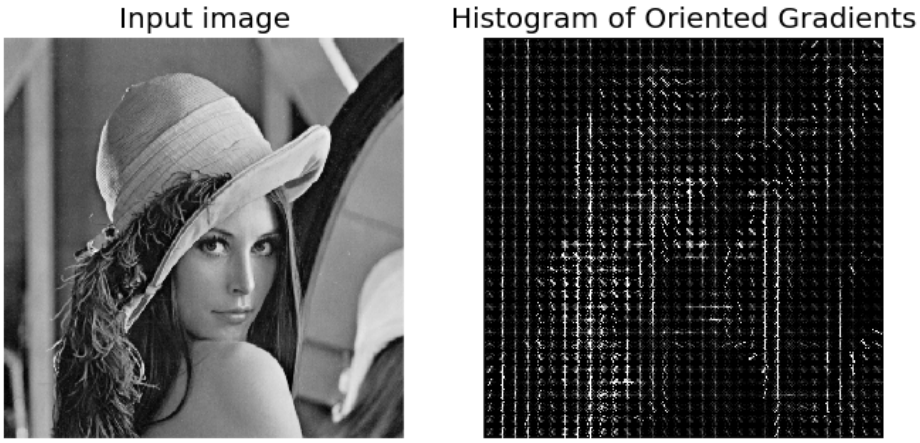

Figure 12: HOG features.
Source: http://sharky93.github.io/docs/gallery/auto_examples/plot_hog.html

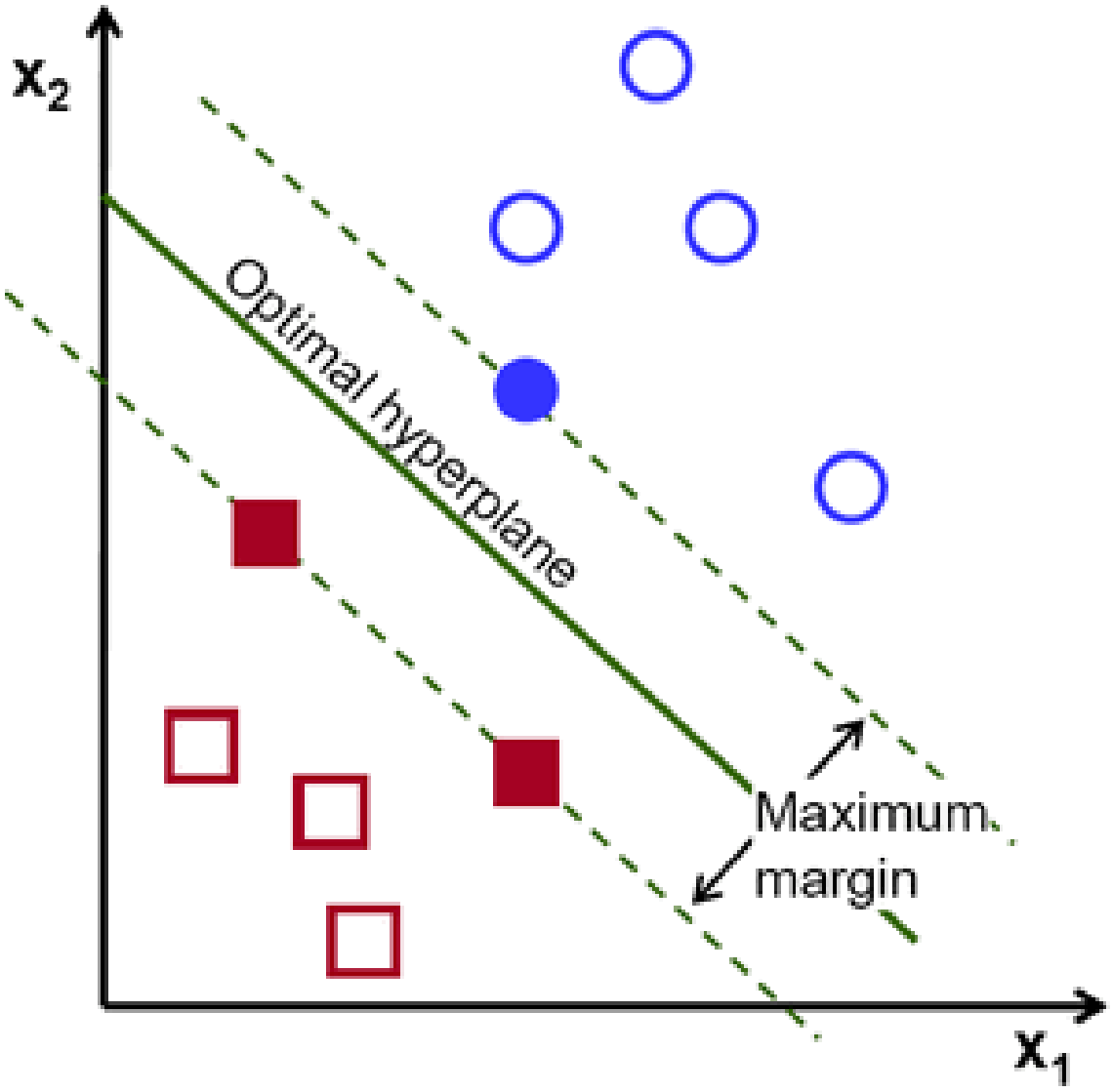

Figure 13: SVM training.
Source: http://docs.opencv.org/2.4/doc/tutorials/ml/introduction_to_svm/introduction_to_svm.html

2.4 Classifier Training

2.4.1 Get Training Data

Positive training images These images are created manually. They are images of cars in different angles and colors. They are obtained from a early version of the simulator but the appeareance has not changed. The simulator provides a screenshot function by pressing "p".

Negative training images I created a function **generateNegativeTrainingData** which produces random images from the simulation environment withour cars. This

JavaScript function can only be accessed via the browser console. The size of generated images in one call is limited by the computers RAM. It takes some time to create thoose images. After the creation they are downloaded as ZIP archive.

2.4.2 Train Cascade Classifier

The cascade classifier for the car detection are trained using a OpenCV tool called *opencv_traincascade* and *opencv_createsamples*. It uses boosting to train the classifiers.

To provide a easier access to thoose tools batch skripts are created. The important parameters for thoose are:

Createsamples:

This tool creates one single file containing all positive samples. The locations of the positive samples have to be stored in a file called info.dat This file contains paths to every single positive image and the object bounding box.

Traincascade:

data The directory where the resulting classifier and intermediate results are stored.

vec The path to the vec file in which the positive samples are stored. This file has to be created using opencv_createsamples.

bg Path to the background file bg.txt which contains a list of all negative samples. IMPORTANT: The background file must contain absolute paths to every single image location.

numPos Number of positive samples.

numNeg Number of negative samples. (Usually much more than positive samples).

numStages Number of cascade stages. In each stage contains weak classifier to create one strong classifier. I archived good results between 20 and 30 stages.

w Width of the positive samples and the detections. (depends on object)

H Height of the positive samples and the detections. (depends on object)

maxFalseAlarmRate since we use weak classifiers this value always should be 0.5 (slightly better than chance)

minHitRate Minimal acceptance ratio off true detections. (usually ¿ 99

featureType HAAR features or LBP features. LBP is much faster!

```
C:\montiarc4-features\features\autonomousdriving\computervision\machineLearning\cascadeClassifierTraining\win64>opencv_t
raincascade -data res/classifier -vec res/trainingData/posSamples.vec -bg res/trainingData/bg.txt -numPos 500 -numNeg 19
230 -numStages 30 -w 50 -h 50 -maxFalseAlarmRate 0.5 -minHitRate 0.999 -precalcValBufSize 3072 -precalcIdxBufSize 3072 -
featureType HAAR
--------------------------------------------------------------------------------
Training parameters are pre-loaded from the parameter file in data folder!
Please empty this folder if you want to use a NEW set of training parameters.
--------------------------------------------------------------------------------
PARAMETERS:
cascadeDirName: res/classifier
vecFileName: res/trainingData/posSamples.vec
bgFileName: res/trainingData/bg.txt
numPos: 500
numNeg: 19230
numStages: 30
precalcValBufSize[Mb] : 3072
precalcIdxBufSize[Mb] : 3072
acceptanceRatioBreakValue : -1
stageType: BOOST
featureType: HAAR
sampleWidth: 50
sampleHeight: 50
boostType: GAB
minHitRate: 0.999
maxFalseAlarmRate: 0.5
weightTrimRate: 0.95
maxDepth: 1
maxWeakCount: 100
mode: BASIC
Number of unique features given windowSize [50,50] : 3024775

Stages 0-1 are loaded

===== TRAINING 2-stage =====
<BEGIN
POS count : consumed   500 : 500
NEG count : acceptanceRatio    19230 : 0.216469
Precalculation time: 72.67
+----+---------+---------+
|  N |    HR   |    FA   |
+----+---------+---------+
|   1|        1|        1|
+----+---------+---------+
|   2|        1|        1|
+----+---------+---------+
|   3|        1|        1|
+----+---------+---------+
|   4|        1|        1|
+----+---------+---------+
|   5|        1|        1|
+----+---------+---------+
|   6|        1|        1|
+----+---------+---------+
|   7|        1| 0.952938|
+----+---------+---------+
|   8|        1| 0.957722|
+----+---------+---------+
|   9|        1| 0.868799|
+----+---------+---------+
```

Figure 14: opencv_traincascade

2.5 Depth estimation

2.5.1 Stereo Camera

To estimate the distance to the object a stereo camera is needed. Three.js provides functions to use a stereo camera. Therfore StereoEffect.js is responsible to render the image in stereo. Unfortunately the original version of StereoEffect.js contains a bug which such that ne depth information where in the image. This bug was fixed in the implementation.

2.5.2 Stereo Reconstruction

For the stereo reconstruction again OpenCV 3.2 is used. It contains a function to calculate the disparity of pixels (distance from the same pixel in the two images). Out of this the depth can be calculated by the baseline of the stereo camera and the focal distance.

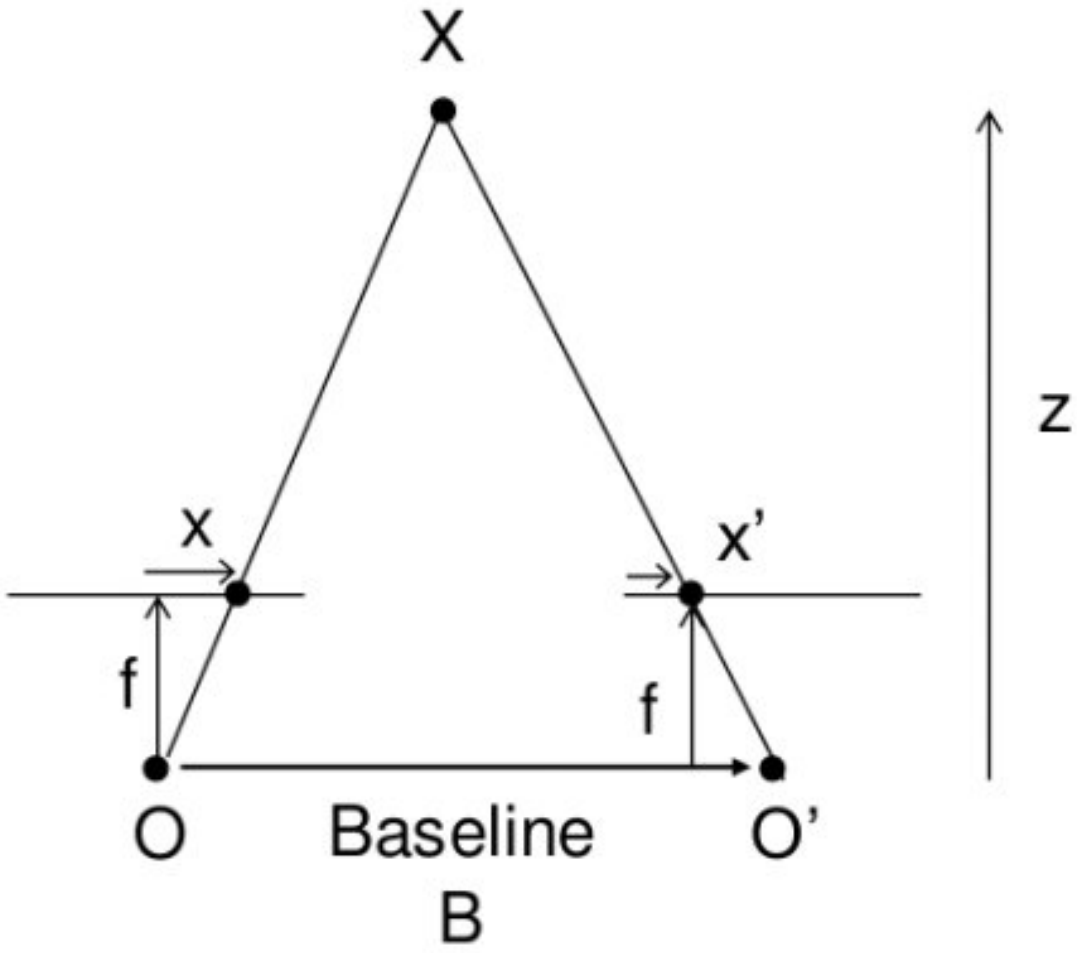

Figure 15: Disparity to depth.
Source: http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_calib3d/py_depthmap/py_depthmap.html

2.6 Tracking

Until here objects are detected in each time step but there exists no relation between the detected objects from the different time steps. Tracking those objects over time can be useful to improve for example the collision avoidance.

The goal of tracking is to associate each detection with previous detections of the same object. Here a simple gating and nearest neighbor assignment approach is chosen This approach consists of two steps: First movement prediction where the next location of the objects is estimated using a kalman filter. The second step is to search for all detections in a gating area and assign the nearest detection to the object (see figure 16).

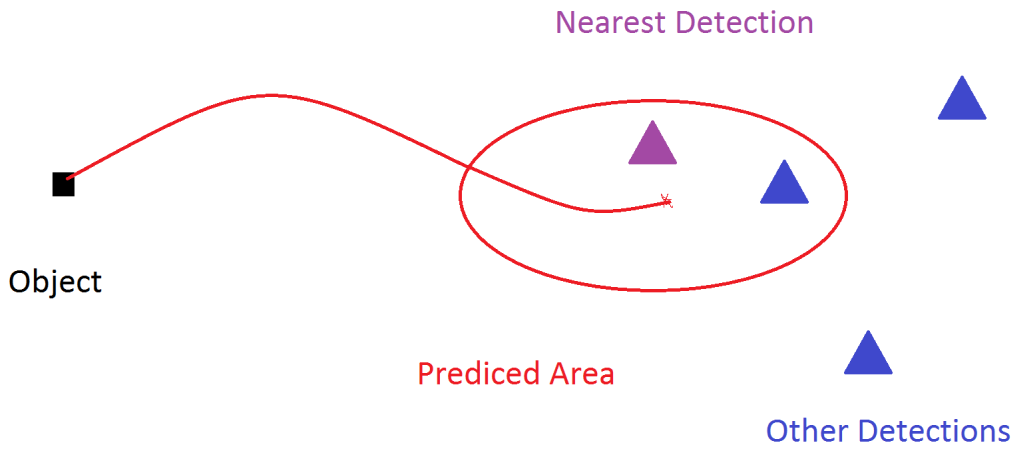

Figure 16: Nearest neighbour gating

This feature can easily be integrated in the existing approach because it uses the detections and further processes them. Responsible for this is the class **OpenCVObjectTracker** (see figure 17).

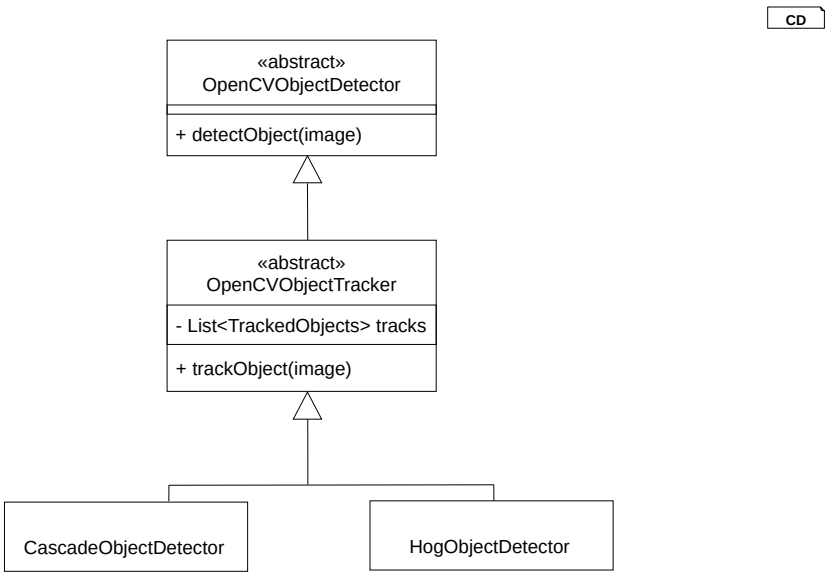

Figure 17: Class diagram tracking

As a result the CarDetector and the PedestrianDetector are also able to track cars and pedestrians.

2.7 Possible Optimizations

Object Detection Method There exist more powerful detectors which have less detection errors than the here used HOG Detector and the Cascade Detector provided by OpenCV. For example one could use integral channel features instead of HAAR features.

Classifier Training This can be optimized by adding more positive and negative samples of the objects.

Tracking The used nearest neighbor gating approach can produce wrong results. This could be replaced by Multi-Hypothesis-Tracking or a image matching approach which calculates the similarities between the detections.

2.8 Remarks

This report is no scientific report. Its purpose is to inform what happens in the object detection procedure. Some images are from the web. The images only show some basic of what happens it does not mean I have implemented all of those features myself.

3 Integration

3.1 Integration of OpenCV

Most computer vision components use OpenCV 3.2 functions. Unfortunately OpenCV is not available as Maven dependency and has to be installed and build at each operating system individually. Nevertheless a platform independent integration of OpenCV would be desirable. Therefore we use a version of OpenCV which is packed by OpenPnP. They provide a simple maven dependency for OpenCV. Additionally it provides the needed binaries for a set of different operating systems including Linux, MacOS and Windows. By this dependency the Java bindings of OpenCV can easily used in the whole project. Only the binaries have to be loaded by calling the following commands before the first usage of OpenCV.

```
nu.pattern.OpenCV.loadShared();
System.loadLibrary(Core.NATIVE_LIBRARY_NAME);
```

3.2 Integration of computer vison components

All computer vision components have to be integrated as sensor in the vehicle. Therefore we created the class **CVSensorSetup**. It contains static methods to add all registered cv components to a vehicle (**addCVSensorsToVehicle**). It needs the CameraSensor and the DataBus explicit as arguments.

This functional module also contains methods to test and demonstrate all cv components with different sequences of test images.