# 

# Contents

[1. Introduction 1](#_Toc328550271)

[1.1 Motivation 1](#_Toc328550272)

[1.2 Project Context 1](#_Toc328550273)

[1.3 Outline 2](#_Toc328550274)

[2. Project Objectives and Specifications 4](#_Toc328550275)

[2.1 Problem Statement 4](#_Toc328550276)

[2.2 Challenges 5](#_Toc328550277)

[3. Bibliographic research 7](#_Toc328550278)

[3.1. Theoretical Background 7](#_Toc328550279)

[3.1.1 Image Processing Techniques 7](#_Toc328550280)

[3.1.2 Background Subtraction 10](#_Toc328550281)

[3.1.3 Linear SVM Classifiers 11](#_Toc328550282)

[3.1.4 Object Representation, Feature Extraction, HOG 12](#_Toc328550283)

[3.1.4 Trackers 13](#_Toc328550284)

[3.2. Related Work 14](#_Toc328550285)

[3.3. Tools 15](#_Toc328550286)

[3.3.1 OpenCV 15](#_Toc328550287)

[3.3.2 SVM Light 15](#_Toc328550288)

[4. Solution Analysis and Architecture 16](#_Toc328550289)

[4.1 Training set acquisition 17](#_Toc328550290)

[4.2 Feature Extraction and Classifier 24](#_Toc328550291)

[4.3 Detector 26](#_Toc328550292)

[4.4 Background Subtractor 27](#_Toc328550293)

[4.3 Tracker 30](#_Toc328550294)

[4.3.1 Kalman-Filter Estimation 31](#_Toc328550295)

[4.3.2 Lucas Kanade Optical Flow, Median Flow Filter 34](#_Toc328550296)

[4.3.3 Merging Track Predictions, Find Detection Match, Merging Detections to Tracks 37](#_Toc328550297)

[4.3.4 Matching methods 38](#_Toc328550298)

[4.3.5 Scoring tracks 40](#_Toc328550299)

[5. Detailed Design and Implementation 41](#_Toc328550300)

[5.1 Language, Technology, Frameworks 41](#_Toc328550301)

[5.2 System packages 42](#_Toc328550302)

[5.2.1 Vehicle Tracking Interfaces (VtInterfaces) 43](#_Toc328550303)

[5.2.2 Vehicle Tracking Implementation (VtImplementation) 45](#_Toc328550304)

[5.2.3 Training Set Acquisition 48](#_Toc328550305)

[5.2.4 Classifier Training package 50](#_Toc328550306)

[5.2.5 Tracking 50](#_Toc328550307)

[6. Testing and Validation 56](#_Toc328550308)

[6.1 SVM Training Error – Precision and Recall 56](#_Toc328550309)

[6.2 LBP-NCC Comparison 57](#_Toc328550310)

[6.3 Examples of tracking errors 59](#_Toc328550311)

[7. User manual 61](#_Toc328550312)

[7.1 Installation Requirements and Guide 61](#_Toc328550313)

[7.2 User Guide 62](#_Toc328550314)

[8. Conclusions 64](#_Toc328550315)

[8.1 Summary 64](#_Toc328550316)

[8.2 Further Development 65](#_Toc328550317)

[Bibliography 66](#_Toc328550318)

[Appendix A - VtInterfaces 68](#_Toc328550319)

[Appendix B - VtImplementation 70](#_Toc328550320)

[Appendix C - DataSetExtraction 74](#_Toc328550321)

[Appendix D - HogFeatureExtraction 78](#_Toc328550322)

[Appendix E - VehicleTracker 80](#_Toc328550323)

# 1. Introduction

## 1.1 Motivation

The problem of traffic management and finding vacant parking spaces in highly urbanized cities is inevitable for drivers and authorities alike. Transportation can be a frustrating experience when spending an unreasonable amount of time at a traffic light, or when searching continuously for a vacant parking space in a neighborhood of interest.

The first attempts of the authorities to solve these problems were to allocate resources, time and money, to build physical infrastructures: roads, intersections, parking spaces to accommodate the increasing number of vehicles and traffic. These solutions only partially solved the problems because people do not have the proper information and the means to acquire it in order to efficiently use the infrastructure, for example to decide an optimal route for their destination in advance, thus ending up in highly loaded highways with no means to escape it.

In this context a solution for acquiring, analyzing and disseminating real time traffic information is a necessity for all transportation stakeholders. Today we already have an enormous capacity and resources of disseminating data to drivers engaged in traffic due to an exponential increase of mobile terminals usage, albeit smartphones or tablets.

The difficulty to address here are the methods on how to intelligently reason and analyze the traffic information acquired, and what are the hardware requirements that will suffice in achieving this because we already know that the amount of processing is immense.

A successful solution would be a system capable of monitoring road segments, with low cost acquisition and processing devices. The final phase in the deployment of such a solution is to integrate all these individual systems providing users with information of traffic status for a specific location or automating the process of finding an optimal path for their destination. Such a system could also have the capacity of finding and indicating vacant parking spaces to users.

## 1.2 Project Context

Computer vision is a field concerned with methods for acquiring, processing, analyzing and reasoning about high-dimensional data from the real world in order to produce semantic information. In the context of this field researchers have elaborated novel methods to duplicate the abilities of human learning and vision and perception.

Recently more commercial interest was given to a broad range of computer vision solutions. Recent development efforts have tried to accommodate computer vision in diversified areas like assisted vehicle driving and navigation, interaction between human and computer, automatic surveillance, controlling processes, automatic inspection, abnormal activity detection in scenes.

Equivalent solutions for analyzing the traffic on road are in-road technologies such as inductive loop detectors or other sensor technologies relying on microwave, infrared, laser, magnetometers. The disadvantage of these approaches is the increased costs of installation and maintenance. In the case of in-road inductive loop detectors traffic must be disrupted and thus causing serious inconveniences. Another disadvantage in case of the specified sensor based technologies is that the installation of such devices relies on road topology. It is difficult of installing such technologies on road regions where vehicles have the behavior of making turns or where there are significant highway corners. Thus the output of these sensors is a poor description of the traffic events. Vision based systems have the capacity of conveying much more information from the scene under surveillance, by observing multiple lanes, interpreting events or extracting discriminative features from passing objects.

One of the first computer vision based systems designed to replace the inductive loop detectors was introduced in America and used localized detection zones, i.e. zones marked on the image where the pixels were monitored. Under a good camera placement assumption (camera placement high above the ground, head-on view) and ideal environmental conditions (clear weather, and absence of shadows) these systems could obtain excellent results. Under non-ideal conditions these systems could record false detections and wrongfully trigger acknowledgements.

In order to overcome the incapability mentioned above we must expand the scope of the system to detect and track vehicles over time (long term trackers) during which we can extract information of interest from the appearance and behavior of objects that the system will use to infer a justifiable and reasonable decision about the nature of the object. By performing these operations we can also enable the inference the scene events: near crashes, abnormal driving patterns, driving over the speed limit. These capabilities are not achievable by simple image processing techniques.

Under the stated motivation, described context and preliminary identified shortcomings we propose a solution that relies on a stack of advanced computer vision and artificial intelligence methods, i.e. algorithms and techniques to acquire, process and reason about the scene of a road segment, for which we want to extract and infer information about the traffic flow.

With the availability of powerful and low-cost computing resources, using computer vision for detection and tracking of vehicles is now feasible for practical applications. Another strong point in preferring a vision based system is the existence of already mounted surveillance cameras, thus reducing time for deployment and cost because it is fully pluggable and compatible with such surveillance systems.

As stated above the two main challenges that this kind of system tackles are detection and tracking. Although considerable effort was performed in the last years to obtain the formulization of these problems and state of the art robustness, we are still very far from matching up to the human visual system and level of understanding.

## 1.3 Outline

This present paper is structured into eight chapters, in which the study domain, the analysis, design and implementation are presented, as well as results, identified challenges and possible further development.

*Chapter 1* identifies the main context of the paper and presents our motivation and advances that people could benefit upon a successful introduction of such a computer vision system, but also the challenges that impaired research activities to formally state the solutions for these approached problems.

*Chapter 2* describes the proposed theme along with clear objectives and specifications. Functional and non-functional requirements of the VDT System (*Vehicle Detection and Tracking System*) are presented here.

*Chapter 3* presents related bibliographic research activities from which this project was inspired. Here we will outline the similarities and differences of our methods as well as advantages disadvantages.

*Chapter 4* illustrates the analysis and theoretical foundations of the implemented application and proposes a high level modular architecture showing how to combine the stack of methods and algorithms to achieve the end objectives.

*Chapter 5* contains the detailed design and implementation, accent here being put on identifying the concerns of the composing modules.

*Chapter 6* presents the methods and difficulties and results in evaluating this system.

*Chapter 7* presents the user manual, i.e. the necessary steps in installing and using the system.

*Chapter 8* aims to summarize the contributions of the paper, while taking a critical look over the results achieved and attempting to describe possible improvements and further development.

# 

# 2. Project Objectives and Specifications

## 2.1 Problem Statement

The scope of this paper includes several fields of study from computer vision and artificial intelligence namely image processing and machine learning methods for object classification, object detection and object tracking. These are all subjects which individually have been the topic of many papers because the problems they present are impairing the computer vision domain for years.

This paper aims to present an effective and simple approach for automating the image dataset extraction which is necessary in all computer vision learning applications, to find and evaluate an efficient method for vehicle classification and detection, and lastly employ these in elaborating and evaluating a new tracking algorithm. Another objective that we propose is to design and formalize a configurable and pluggable architecture aimed for object tracking based on which we can evaluate the performance of other implementations of the proposed system modules.

We will employ the specified objectives in the context of evaluating the traffic flow on a specified road segment to estimate the number of cars that pass during the time frame of the observation. We will describe and implement a system whose primary input is a video sequence taken from a single acquisition device and outputs the vehicle objects in the process of detection and tracking in an annotated window while also keeping count of how many vehicle objects it has registered and scored as belonging to this class.

*Functional Requirements*

1. The main input of the system is the video sequence from a single acquisition device*, a video camera*. It is not within the scope of this paper to consider a 3D reconstruction of the scene using more cameras. While this is a complex approach, it can be a method which can gain discriminative information about the depth of every pixel.
2. We must achieve vehicle tracking by only processing *2D image sequences* , the projections of the scene on the camera plane, without the aid of other sensors
3. The mountable position of the camera is also constrained; specifically the camera must be placed in a high viewing point for reasons detailed in later chapters.
4. A *low end camera* must be used as a video acquisition device
5. The surveillanced scene must exhibit a fairly linear pattern of motion and small variations of scale and size around a median scale and size of vehicles.

*Non-Functional Requirements*

1. *Performance*. The system must be delivered the output *real-time* so that it can be properly interfaced with a video acquisition device. A frame rate around 10 fps will suffice for this stage of development. We will consider porting some processing on the Graphical Processing Unit if necessary.
2. *Scalability*. By increasing the hardware capacity the system should be able to process scenes in increasing complexity and configurations while maintaining or improving performance.
3. *Configurability*. The system has to expose the ability of configuring existing modules that address well defines concerns or give possibility of replacing the modules with other components that conform to the same contracts.

As stated above, in order to achieve these results we must employ of a stack of computer vision and artificial intelligence methods. To achieve these objectives we should have understanding of how these are formalized in the literature. In what follows we give brief description of the problems we are tackling to better understand the scope of the project.

*Machine Learning* is integral to object recognition and a big part of computer vision, but it’s a field worthy of its own book. The goal of machine learning(ML) Machine learning turns data into information by extracting rules or patterns from that data. After learning from a collection of data, we want a machine to be able to answer questions about the data: What other data is most similar to this data? Is there a car in the image?

In the context of machine learning we use *classifiers* which are methods for determining the likely class of an unknown object or event, based on a number of instances from each class, known as the training set. There are a number of these methods which differ in approach such as Support Vector Machines (SVM), Boosting, Random Trees, Neural Networks, K-nearest neighbors. We are using a Support Vector Machine because it has received a lot of attention and good results in recent years.

*Detection* is a mechanism which every tracking method requires for indication of object appears and reappears in the video, supplying information of locality and dimensions or bounds. Simple procedures to perform detection rely on information extracted from a single frame*.* Most often we need temporal information resulted from operations of frame differencing, which identifies moving regions in consecutive frames. Several common detection methods rely on point detectors, which find interest points holding discriminative texture information, background subtraction by building a model of the background specifying which regions of the scene are static, segmentation by partitioning the image into similar regions. Another advanced approach in detection is the use of a classifier, which has leant different object views automatically from a set of examples, called supervised learning. In our project we used different methods of background subtraction together with supervised learning.

*Tracking*, in its simplest form, can be defined as the problem of estimating and generating the trajectory of an object over time by locating its position in every frame of the video. Object tracker may also provide the complete region in the image that is occupied by the object at every time instant. In other words, a tracker assigns consistent labels to the tracked objects in different frames of a video. Possible object regions in every frame are obtained by means of an object detection algorithm, and then the tracker matches objects across frames. Additionally tracking can provide object-specific information, such as orientation, area or contour of an object.

## 2.2 Challenges

#### 2.2.1 Image dataset acquisition

In general, all machine learning algorithms take as input thousands of data vectors made up of many features, where the number of features might also reach thousands.

The first problem encountered is how to collect the images that make up the dataset. Do you search for online datasets or do you collect it from a security camera and what kind of information do you capture: static or movement. We must end up with thousands of images.

#### 2.2.2 Labeling the dataset

Our task is to recognize certain type of objects, vehicles. Another problem encountered is how to label training data that falls into positive (there is a vehicle in the scene) and negative (no cars) cases. We wish to do this in an automatic, and minimize the human interaction in this process.

#### 2.2.3 Features, Object Representation

It is unavoidable that cars will appear at different scales: their image may consist of just a few pixels, or may be covering the whole screen. Even worse, vehicles will often be occluded: a vehicle passing behind a tree, a man passing in front of the car, or vehicles occluding each other. You need to capture the variations in the data: different views of vehicles, different lightings, weather conditions, shadows, and so on.

After labeling the data, we must decide which features to extract from the objects. Again, you must know what you are after. In general, we must find features that express some invariance in the objects.

Detecting cars is a considerably more difficult problem than textured objects which have simple, semi-rigid structure, where the localization of components does not vary much between samples. Cars have a semi-rigid structure as well, but that structure will vary more between samples, because their shapes and configurations have been designed with product differentiation in mind.

Besides the intra-class variations due to color, shape, and ornamentation, there are other issues that complicate car detection. First, we wish to be able to detect a car under small rotation variations. Compared to the other object classes, a car also lacks texture and has a highly reflective surface.

#### 2.2.4 Tracking

Tracking is also plagued with the following misfortunes: loss of information caused by projection of the 3D world on a 2D image, noise in images, complex object motion, non-rigid or articulated nature of objects, partial and full object occlusions, complex object shapes, scene illumination changes

#### 2.2.5 Performance, Hardware Requirements

Considering that every frame from the video sequence will need to be processed individually with image processing algorithms: background subtraction, thresholding, scanned for vehicle occurrences, perform track evaluation and estimation; we start to worry about performance and must find ways to bring the performance of the system to real-time throughput. Hardware capability is a decisive factor that influences performance of the system. An important issue here is also the quality and size of the acquired images: while having high quality pictures at high resolutions it more reliable to perform object representation and discrimination, it will drastically decrease performance, while having low resolution images will increase performance but will result in loss of information.

# 

# 3. Bibliographic research

## 3.1. Theoretical Background

### 3.1.1 Image Processing Techniques

Over the course of this chapter we will describe and define the basic notions and the whole stack algorithms and techniques from the simplest to the most complex that we will refer within the rest of this paper, as well as some state of the art solutions that inspired this paper. You will find that there are a variety of approaches with advantages and disadvantages which we will try to highlight and also mention which are the similarities and differences of our paper based with each of them. Let us begin by defining the building blocks of this solution.

#### 3.1.1.1 Grayscale Images

A grayscale image has the property that the color at the level of each pixel is represented by a single value, a shade of gray, varying from black, the lowest value, to white, the highest value. The reason for which these versions of images are widely used is that they carry less information but are discriminative enough to support high level processing. In practice a grayscale image is one in which the red, green and blue components all have equal intensity in RGB space, and thus is only necessary to specify a single value which is common for the three RGB channels. It is thus intuitive that we can convert any color image into an equivalent grayscale image by replacing the value of each pixel with the arithmetic mean of the three color channels.

Figure 3.1 Color image Figure 3.2 Grayscale image

#### 3.1.1.2 Thresholding

Thresholding is a simple but effective way to separate objects of interest from the background based on the values of the pixels. Each pixel is individually labeled as belonging to the objects if its value is higher than a predetermined value or belonging to the background otherwise. There are many variants of thresholding; the method mentioned bears the name threshold above. Another variant, called threshold below, labels the pixels as object pixels if they are within the lower range, threshold inside, labels the object pixels if they have their value within a bounded range. Another variant is called adaptive thresholding in which different thresholds are used for different regions of the image. Figure 3.3 is a sample input image of the threshold above operation and figure 3.4 is the output.



Figure 3.3 Original image Figure 3.4 Example of thresholded image

#### 3.1.1.3 Morphological Operators

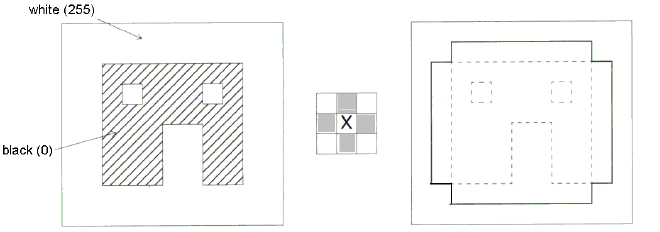
Morphological defines a series of operators which transform an image by probing it with a predefined shape element, named structuring element. The way this shape element intersects the neighborhood of a pixel determines the result of the operation. ([1])

A structuring element, which can be depicted in Figure 3.5.c, is defined as a grid or configuration of pixels on which an origin is defined, also called anchor point. When the origin of the structuring element is aligned with a given pixel, its intersection with the image defines a set of pixels on which a particular morphological operation is applied.

There are two common operators in the context of morphology, erosion and dilation. In the case of erosion, the pixel overlapped by the origin of the structuring element is replaced by the minimum from the pixel set overlapped by the structuring element. In the case of dilation, it is the inverse operation, the maximum is extracted from the pixel set overlapped by the structuring element and is placed into the pixel overlapped by the origin of the structuring element.

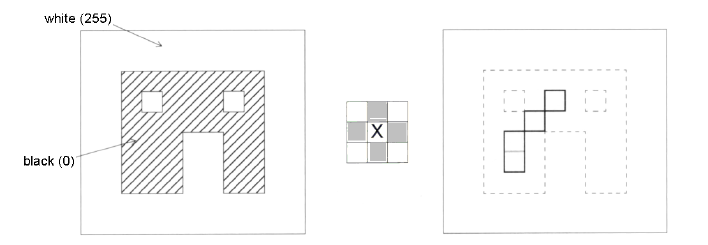
Based on these two operators are defined two operations, closing and opening. Closing is defined as the erosion of the dilation of an image and it is used to fill the holes and close the gaps in an image. Opening is defined as the dilation of the erosion of an image and is used to eliminate pixels in regions that are too small to contain the structuring element. We are using the opening operation as a method to discard noise and to remove remote regions of pixels that could interfere with processing.

Figure 3.5 is an illustration of the dilation process, on the left there is a sample input image, in the middle is the structuring element marked with its origin, and on the right is the result of the dilation operation. You can observe that the background pixels that fell in the origin of the structuring element near to the object pixels, having higher values, as it was swept across the image have been replaced by the object pixels, and a particular effect is the linking of near object pixel regions and closing small holes. Figure 3.6 is an illustration of the erosion operation. You can observe that object pixels that fell within the structuring’s element origin near the structuring’s element range that also captured the background have been replaced by background pixels. An effect of this operation is the removal of links between loosely connected regions and the reducing the size of object pixels or completely removing them if they do not fit the structuring element.



a. Original Image b. Structural c. Image after dilation; original element; x=origin image in dashes

Figure 3.5 Source: [24], Illustration of the dilation process



a. Original Image b. Structural c. Image after erosion; original element; x=origin image in dashes

Figure 3.6 Source: [24], Illustration of the erosion process

#### 3.1.1.4 Geometrical Properties

*Area*

The area of an object is measured in pixels and indicates the relative size of the object. In general we use this property to discard regions of pixels which are too small that they do not present further interest in processing or which are too big which is an indication that multiple objects might be contained in the region.

*Contour*

A border tracing algorithm is used to extract the contours of an object from an image. When applying this algorithm it is assumed that the image regions have been previously labeled. We will use this property to reason on the form of objects.

*Perimeter*

The perimeter of the object helps us determine the position of the object in space and provides information about the shape of the object. The perimeter can be computed by counting the number of pixels on the contour.

*Circularity*

The circularity or thinness ration is a property for evaluating how round the object is. The value of the thinness ratio also offers information on how regular an object is. It can be a method for differentiating between objects that have a regular contour and objects of irregular contours.

*Convex Hull*

In mathematics, the convex hull or convex envelope for a set *X* of points in the Euclidean plane is the minimal convex set containing *X*. On the left of figure 3.7 is an example of a vehicle’s shape obtained from background subtraction and on the right is the convex hull.

convex_hull_original convex_hull_processed

Figure 3.7 On the left - the original set of points,

on the right - the convex hull of the respective set

### 3.1.2 Background Subtraction

Background subtraction is a common approach for identifying motion, and involves creating a background model to represent the scene with no foreground objects.

In our case it is necessary to identify the pixels corresponding to regions of projected vehicles onto the image plane. Accurate identiﬁcation of these regions will serve as input for tracking.

A background subtraction method must address many challenges. Passing vehicles must be segmented under continuous environmental change (lighting conditions, weather conditions) and noise.

The idea for identifying the groups of pixels belonging to moving objects is based on the differentiation of the current image from the background model.

There are several methods of performing background subtraction out of which we will use two: Running Average and Mixture of Gaussian.

#### 3.1.2.1 Running Gaussian Average

This approach dynamically builds the background model by regularly updating it. This can be accomplished by computing what is called a running average. The value of each pixel in the background model is equal to a weighted average of the value from the background model in the same location at time *t-1* and the value we currently have on the frame at time *t*. This method is one of the simplest techniques of background subtraction and has the advantage of low memory usage and short running time. In the following chapter we will detail the formula and the context in which we used it.

#### 3.1.2.2 Mixture of Gaussians (MOG)

This approach constructs the background model by keeping one or more Gaussian distributions at the level of each pixel for the most frequent observed values. When the color of a pixel does not vary around the mean value of one of the maintained Gaussians models then it is excluded as being part of the background model and is considered a foreground pixel, and when it varies around the mean of a maintained Gaussian model then the pixel is considered to be part of the background and the distribution is updated accordingly.

#### 3.1.2.3 Shadow removal

Although it is beyond the scope of the project, a shadow removal technique can be used to further obtain a finer filtering of foreground objects. If we consider working with a color model that separates chromatic and brightness components we could determine the shadows by comparing a non-background pixel against the current background components. If the difference of chromatic and brightness components are within some thresholds, the pixel is considered as a shadow, and thus is part of the background.

### 3.1.3 Linear SVM Classifiers

In the past years SVMs have widely been used as a classification method and were capable of delivering state of the art results, sometimes better than other approved methods, such as neural networks or maximum likelihood.

Given sets of samples belonging to different classes the SVM method relies on finding the optimal separating hyperplane between these classes. The separating hyperplane is computed relative to the samples that lie on the boundary of the classes, these are called support vectors. Training samples other than support vectors are of no use in classifications and thus can be discarded.

A complete formulation of Support Vector Machines can be found in [2] and [3]. Let us consider a supervised binary classification problem. The training data are represented by {xi, yi}, i = 1, 2, …, N, and yi ∈ {-1, +1}, where N is the number of training samples, yi=+1 for class ω1 and yi=-1 for class ω2. To make things simple let us suppose the two classes are linearly separable. This means that it is possible to find at least one hyperplane defined by a vector *w* (which is normal to the hyperplane) with a bias *b* (which is an indication of the displacement from the origin), which can separate the classes without error. Figure 3.8 is an illustration of the training data depicted in a two dimensional space, and the optimal separating hyperplane which in our chosen dimensions is a line.

The equation of the hyperplane has the following form:

*f* (*x*) = *w⋅ x - b* = 0

The support vector will satisfy the following constraints:

*f*(*x*) = *w⋅ x - b* = +1, for *x* from ω*1* class

*f*(*x*) = *w⋅ x - b* = -1, for *x* from ω*2* class

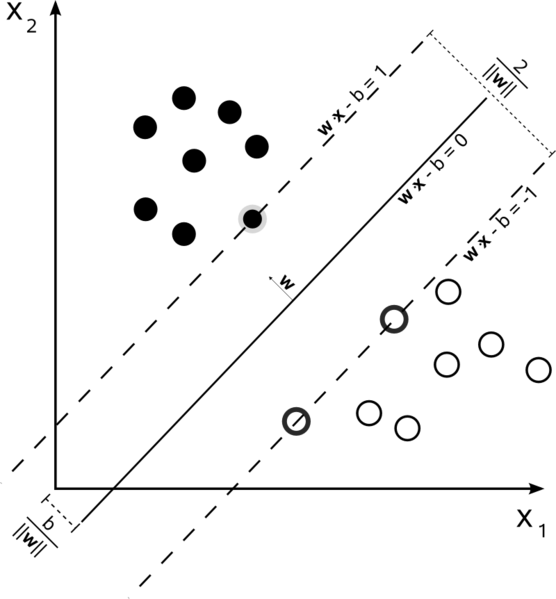


Figure 3.8 The picture depicts the training samples in a two dimensional space, denoted *x1* and *x2*, black samples correspond to ω*1* class and the white samples to ω2 class.

It is beyond of the scope of this paper to offer the full demonstration of how to deduct the expression for *w* and *b*, but we mention that this function can determine the class label of a tested sample by evaluating its sign in sampled point.

### 3.1.4 Object Representation, Feature Extraction, HOG

In a tracking scenario, an object can be defined as anything that is of interest for further analysis. There are multiple methods based on which you can rely on describing an object, but strongly depends on the domain of the application. The possibilities are based on point descriptors, primitive geometric shapes, object silhouette and contour, skeletal models, templates. In later chapters we will have the motivation of using a simple object representation method based on the circularity of the convex hull.

The seconds object representation technique that we will be using is called Histogram of Oriented Gradients (HOG). We will give a brief description of it because it lies at the heart of the classification and detection problem.

The essential thought behind the Histogram of Oriented Gradient descriptors is that local object appearance and shape within an image can be described by the distribution of intensity gradients or edge directions. The implementation of these descriptors can be achieved by dividing the image into small connected regions, called cells, and for each cell compiling a histogram of gradient directions or edge orientations for the pixels within the cell. The combination of these histograms then represents the descriptor. For improved accuracy, the local histograms can be contrast-normalized by calculating a measure of the intensity across a larger region of the image, called a block, and then using this value to normalize all cells within the block. This normalization results in better invariance to changes in illumination or shadowing.

The HOG descriptor maintains a few key advantages over other descriptor methods. Since the HOG descriptor operates on localized cells, the method exposes invariance to geometric transformations, except for object orientation. Such changes would only appear in larger spatial regions.

### 3.1.4 Trackers

#### 3.1.4.1 Lucas-Kanade Point Tracker

The Lucas-Kanade method is a well appreciated method for estimating the optical flow developed by Bruce D. Lucas and Takeo Kanade. It assumes that the flow is essentially constant in a local neighborhood of the pixel under consideration, and solves the basic optical flow equations for all the pixels in that neighborhood.

By extracting information from the neighborhood of the pixel of interest it achieves more tolerance and robustness to image noise than other methods for tracking points. The disadvantage relying only on local information is that it cannot deduct the optic flow from the interior of uniform regions of the image, regions within which color is fairly constant. Figure 3.9 is an example of paths constructed by tracking points with the Lucas-Kanade Point Tracker.

#### 3.1.4.2 Kalman Filter

As explained in [14], the basic idea behind the Kalman filter is that it is possible, given a history of measurements of a system, to build a model for the state of the system that maximizes the a posteriori probability of those previous measurements. It means that the new model we construct after making a measurement, taking into account both our previous model with its uncertainty and the new measurement with its uncertainty, is the model that has the highest probability of being correct.

The task is divided into two phases. In the first phase, typically called the *prediction phase*, we use information learnt in the past, from previous measurements, to further refine our model for what the next location of the vehicle will be. In the second phase, the *correction phase*, we make a measurement and then compare that measurement with the predictions based on our previous measurements (i.e. our model).



Figure 3.9 Source: [1], Lucas-Kanade Point Tracker example

## 3.2. Related Work

Throughout the course of this section we will analyze some of the most relevant papers that inspired the design and development of our solution.

The goal of the first 3 chapters of [4] is to construct a hierarchical car detector by automatically learning cars specific features and then learning a combination classifier. First they extract SIFT key points from car images in our training set. Each key point is a 128-dimensional feature vector representing gradient orientations of a patch centered in the key point. Key points collected from all training images are clustered by the K-Means clustering algorithm. The assumption is that the centroids of the clusters will be representatives for the most frequent groups of key points, i.e. key points that have a high probability of belonging to the vehicles. In order to train an SVM classifier, they convert the image into a fixed-length feature vector which represents the distances between clusters from the training data to the extracted key points from the image.

In [5] they start with a similar approach as in [6] concerning the detection of vehicles. A background subtraction technique is used to segment the moving vehicles from the static background. The next task is to determine which blobs contain a single vehicle or more occluded vehicles. This information about blobs is deducted from simple morphological characteristics: the existence or more colors, solidity, computed as the area of the blob divided by the area of the convex hull, orientation and eccentricity which we also use. Once the suspected blobs which are thought to contain more vehicles are separated, the next step is to determine which regions in these blobs correspond to different vehicles. The idea is to find object moving differently by hierarchically clustering the motion vectors associated with extracted SIFT feature points from the blob over a number of frames.

The works in [6] closely resemble the approaches that we also took in the overall design and implementation. They collect their training set by filtering blobs, obtained after background subtraction, that pass through a given region of interest. After obtaining the dataset they train an SVM with HOG features. The detection is done by running the SVM classifier on the blobs. Additional processing is performed when detecting suspicious blobs i.e. blobs which have the area bigger than an expected threshold and are suspected have contain overlapping and occluded vehicles. The detection step differs from ours because we run the classifier in a sliding window manner over the whole frame. Tracking is performed by evaluating a constant acceleration dynamic model. This partially resembles the approach that we implemented as we used a Kalman Filter to evaluate the same dynamic model but also used a Point Tracker, the Lucas-Kanade Optical Flow.

The authors of [7], [8] and [9] propose a formulization to the problem of real-time tracking of objects in video streams. The algorithm is named TLD, it is described to be a long term tracker for single objects, being able to dynamically enlarge and prune the model by using good and false detections until it converges. The scope of this project differs from ours in that they perform tracking of a single object of interest that must be defined by a bounding box in the starting frame, while we need to perform tracking of a class of objects. Our interest was captured by the method of performing the tracking. They use the Lucas-Kanade Optical Flow to estimate the scale and position of the bounding box around the object, they call it the Median Filter. This is the approach that we have adopted in using the Lucas-Kanade Point Tracker.

In [10] they propose a system which is deployed in a autonomous driving context and consider to detect two types of traffic patterns. The first type of traffic is generated from vehicles travelling in the opposite direction and the second type from cars travelling the same direction. They detect incoming traffic patterns by finding corner points from one frame and tracking them into the next frame using the Optical flow, and cluster these points into a rectangle based on the location and direction of movement between the two frames. They argue that this method is not appropriate for vehicle having a traffic pattern in the same direction because they do not have salient movement in most cases, and use a Haar-like feature detector trained on rear views of the vehicles.

## 3.3. Tools

### 3.3.1 OpenCV

OpenCV (*Open Source Computer Vision Library*) is a BSD free computer vision programming library equipped with a broad range of algorithms to support real-time image processing on the CPU and GPU.

OpenCV application areas include:

* 2D and 3D feature toolkits
* Facial recognition system
* Human–computer interaction (HCI)
* Mobile robotics
* Motion understanding
* Object identification
* Segmentation and Recognition
* Motion tracking

### 3.3.2 SVM Light

SVM is an optimized implementation of Vapnik's Support Vector Machine for the problem of pattern recognition, for the problem of regression, and for the problem of learning a ranking function.

# 4. Solution Analysis and Architecture

As presented in earlier chapters, the main contributions and final goal of this thesis concerns vehicle tracking which and the possibility of it being deployed from a broad range of locations under little placement constraints. This chapter will clarify and reason about the details of the presented methods from chapter three and their scope within the proposed architecture. We will logically deduct the steps performed for the implementation of the system. Figure 4.1 describes the general conceptual architecture of vehicle tracking system in which we show the flow of data between the composing modules, dependencies and services provided by the components to other components.



Figure 4.1 Conceptual architecture of vehicle tracking system

You will find out that the architecture that we propose for our solution closely resembles the trends also approached in the works of [5] and [6]. Our change in the architecture was purely motivated by performance reasons, and consists of the tracker module simultaneously consuming the outputs from the Detector and Background Subtractor module. We will detail later in chapter 5 where we discuss the dependencies of deploying these component on various hardware and software platforms.

Next we give a brief description of the basic building blocks that compose the architecture and the tasks that they play in achieving the final goal:

* The video camera is the primary input for the system, responsible for interfacing the system with the real world by capturing video frames of the surveillance scene.
* The Training Set Acquisition module is responsible for building the Image Training Set that is later used in extracting vehicle specific patterns with which the Classifier is trained.
* The Image Training Set is just a container for a collection of positive training samples, images that depict single vehicles in their area of concern, and negative training samples, images that are suppose not to contain any vehicles.
* The Classifier has the capacity to discriminate between patterns of vehicles and non-vehicles on an entire image, by learning to distinguish them based on the derivated features as a result from processing the training set.
* The Detector is also fed with incoming frames from the acquisition device and has the responsibility of communicating with the classifier and search the video frames for the location and size of vehicles. We call *a detection* a bounding box on the input frame which centrally encapsulates a vehicle.
* The Background Subtractor module is responsible with performing motion segmentation, i.e. identifying the moving regions, in the frames whether they are pedestrians or vehicles. The Background Subtractor module will provide the detector a foreground mask in which the value of each pixel will indicate if the corresponding pixel from the frame contains the projection of a moving object, or any kind of entity.
* The Tracker consumes detections provided by the Detector and the foreground mask provided by the Background Subtractor and performs additional processing using mechanisms to construct the tracks of moving vehicles over time and also validates the consistency of being generated by real vehicles.

Now that we have a brief description of what each module is responsible for performing we will dive into the algorithmically details of the architecture and decompose each module as they are scheduled to perform their work in the general process of the system.

## 4.1 Training set acquisition

In general, all machine learning algorithms take as input thousands of data vectors made up of many features, where the number of features might also reach thousands.

The first problem encountered is how to collect the images that make up the dataset. Do you search for online datasets or do you collect it from a security camera and what kind of information do you capture: static or movement. We must end up with thousands of images.

But our job is to recognize a certain type of object, vehicles showing in specific image regions. The second problem encountered is how to label training data that falls into positive (there is a vehicle in the scene) and negative (no vehicles) cases. We must end up with thousands of images, how do we label that?

You will soon realize that vehicles appear at different scales, sizes, orientations, design, their image may consist of just a few pixels, or may cover whole screen. Vehicles will often be occluded: a vehicle passing behind a tree, a man passing in front of the car, or vehicles occluding each other.

An algorithm that recognizes vehicles from ground level might fail when applied on top views of vehicles. We need to capture the slight variations in the data: different views of vehicles, different lightings, weather conditions, shadows, different car producer designs.

The training set acquisition module has the objective of addressing the problem of capturing and labeling the whole training dataset. The proposed method for acquiring the training set has the advantage of gathering images offering meaningful patterns for vehicles that are relative to the scene and locality of our acquisition device and also variations in the data, mentioned above.

For example, if we were to deploy this system in two different locations, one filming vehicles close and frontally, and another filming vehicles from far away and sideways, we would expect that this module to collect front facing pictures of cars, in the first case, and sideways pictures of cars, in the second. Moreover the vehicles in one database will be of similar scale and capture only specific variations of orientation that the scene permits: cars having normal orientation in the direction the road, cars engaged in overtaking. Figure 4.2 is the expansion of the Training Set Acquisition module from the Conceptual architecture depicted in Figure 4.1. The figure depicts the internal processing modules and stages performed on the input frame in order to obtain the cropped out vehicles (positive samples) and also negative samples which are saved in the Image Training Set, as the last operation. As you can see, the architecture follows a simple pipeline model, each module taking as input the processed output of the previous module. Let us analyze what are the objectives of each module and what operations and algorithms engaged in fulfilling their purpose.



Figure 4.2 Detailed Training Set Acquisition module

#### 4.1.1 Video Camera

The video camera acts is the acquisition device which captures frames from the observed scene. A discussion is here necessary based on one constraint that we have imposed on our system, namely we have imposed that the mounting point of the camera should be elevated from the ground providing top-views of the passing vehicles.

Because at this stage of processing we are not interested in the performance of the system but only in acquiring quality training samples, this imposed constraint helps us in further processing:

* We minimize the occlusion regions between vehicles at projection of the scene onto the image plane thus increasing the efficiency of extraction, acquiring thousands of samples
* It captures the median orientation and position that vehicles exhibit as they pass through the scene thus the training samples are specially constructed having in mind the locality of the scene and its perspective projection. Figure 4.3 is a sample frame from our video illustrating that your scene has a median vehicle orientation and scale.



Figure 4.3 Sample frame

#### **4.1**.2 Frame cropping

**Figure 4.3 also demonstrates the perspective effect of projecting the scene onto the image plane, vehicles closer to the camera will appear larger than the ones farther away, another effect is the and also introduces significant occlusions. In [6] the region closer to the camera is called *entrance region*, and in this region occlusions between vehicles are minimized.**

**Also in [6] they automate the process of finding this region by running Canny Edge Detection algorithm to and then a Hough line transform to identify the orientation of the highway. Our approach is different in that we demand the user to select to select with the mouse a region of interest in which he expects to find minimal occlusion between vehicles and in which vehicles appear to have a median scale within their passing through the video. Once the region of interest is selected further processing will be carried on only this region within the frames.**

**The constraints for selecting this region are as follows:**

* **It should minimize the occlusion between vehicles**
* **The vehicles should appear to have a median scale and median orientation that these scene exposes**

**The height of the region of interest should be slightly bigger than the height of vehicles. This information will be later used to filter the detected blobs. An example on how to select such a region of interest is illustrated in figure 4.4.**



**Figure 4.4 Depicts selection of the ROI (region of interest)**

#### **4.1.3 Background Subtractor**

**The Background Subtractor module has the responsibility of learning the background model of the region of interest and output the foreground mask, which is a binary image segmenting moving parts or regions of region of interest from the static regions, the background.**

**In order to learn the background model several frames in the beginning of the video are reserved for the construction of the background model, after which the foreground mask is obtained by differentiating the background model with each incoming frame, additionally the model has to be continuously updated in order for it integrate slow varying changes, like the variation of light.**

**The quality of the foreground mask will affect not only the blob detection later in processing, but also the accuracy of the classifier and because that at this stage of training we are not concerned with real-time processing we choose a high performing algorithm for this task, the MOG (Mixture of Gaussian).**

**In the approach of the Mixture of Gaussian, each pixel history is modeled by a mixture or linear combination of *K* Gaussian distributions, where a pixel history is the sequence of pixel values over the last *t* frames. The probability of observing a pixel value as a background value is given by evaluating the expression:**

**Notations:**

**When updating the pixel history with a new pixel value its *K* Gaussian mixtures are evaluated. If it does not have a high enough value against any Gaussian then the distribution with the lowest cost will be replaced with a new one and also distribution means and covariance matrices are updated.**

**The background model for each pixel is chosen to be composed of *B* most reliable and least variance distributions. When evaluating a new pixel value *X* which gives a high probability within these *B* distributions then the pixel value is considered to be a background pixel.**

**In this manner the MOG is continually adapting to small changes of lighting and weather. We also observed that the foreground masks resulting from this procedure are of high quality and are robust to noise and very small camera movements which we have experienced. In figure 4.5 we show the constructed background model, and the extracted foreground mask for two random frames.**







**Figure 4.5 The first and second columns represent separately the region of interest, background model, and foreground mask from two random frames.**

#### **4.1.4 Blob Detector**

**Having the foreground mask the Blob Detector’s job is to quickly find the blobs, the connected components on the foreground masks. It does this by applying a contour tracking algorithm on the foreground mask following the border white object pixels.**

**Additionally this component also performs the following tasks:**

* **It discards blobs which have the width, height and area less than predefined thresholds. In our application the thresholds for width and height are 5 and for the area the threshold is 25. This filtering step will most likely small blobs which are not of interest resulting from noise of small object movements, example: leaf movements in trees, birds flying.**
* **It also discards blobs that are on or very close to the region of interest’s border. The reason for applying this filtering step is that later compute a bounding box around the blob and we want this bounding box to be contained in the region of interest. Another reason for this operation is that we do not want for a vehicle to be saved too many times in the training set and thus we ignore it when entering and exiting the region of interest.**

#### **4.1.4 Blob Filtering**

**The Blob Filtering module performs additional filtering functions and lastly saves positive samples containing vehicles and negative samples which are preferred not to contain vehicles in the Training Set. This module’s goal is to make an educated and simple decision about the incoming blobs from the Bob Detector module to decide which of them are masks generated by moving vehicles.**

**The decision to classify the blobs as vehicles is done based on a feature of shape, namely the *circularity* of the *convex hull*. In the first iterations of development this module was missing and we were collecting elongated objects mainly because we were experiencing slight camera movements. Figure 4.6 shows that street markings were saved as the background subtraction was generating blobs under camera movement.**

Description: E:\Academic\Licence\lic_doc\pics\chapter 4\bad_blob3.png Description: E:\Academic\Licence\lic_doc\pics\chapter 4\bad_blob1.png 

Figure 4.6 Bad images saved in the absence

of the Blob Filtering Module.

**It was a necessity to use a discriminative feature that would correctly classify cars at this stage, unless we were to manually go over all the generated samples again and hand-pick them.**

**The idea of using the eccentricity of the convex hull is also mentioned in [5]. They argued that a vehicle is roughly a convex object, and the silhouette is very close to its convex hull. When two vehicles tend to occlude themselves the blob containing them generally does not have a convex shape. With this assumption in mind we can further filter passing blobs and discard joint vehicles appearing as one blob.**

**At this stage of processing we are able the robustly filter and discard the blobs which have a high probability of not belonging to the vehicle class. The last function that this module performs is to save the detected blob as a positive sample in the training set. This step is performed by centering a cropping rectangle over the center of gravity of the detected blob. The cropping rectangle’s size is computed relative to the size of the region of interest, namely the height of the cropping rectangle in our case is the largest multiple of 8 which is also smaller than the height of the region of interest, and we compute the width based on an empirically selected aspect ratio which in our case is 1. For our tested video a good cropping size for the saved patches was 64x64. Figure 4.7 illustrates a few cropped frames under the region of interest selected and the detected vehicle blobs passing.**

Figure 4.7 Detected blobs

Once we have saved the cropped images as positive samples in the training set, the next question is from where we obtain the images serving as negative samples. In [6] they propose the idea of saving the negative samples relative to the location of the blob detections. Specifically, they select rectangular patches centered around the corners and edge midpoints.

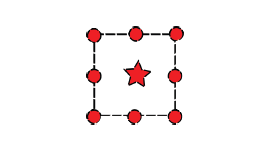


Figure 4.8 Source: [6], Relative position of positive and negative training samples.

The bullets are the centers of the negative patches placed around the corners and edge midpoints of the detection.

**Another strategy of choosing the negative samples would be to select them randomly over the frame but there is the risk that one of them would end up containing a vehicle in the center which will have a negative impact on the training of every type of classifier.**

**The authors of [6] point out that this approach offers a great advantage for the training of the classifier: it is not just trained with plain and random clutter, but with hard examples generally containing margins and portions of vehicles. Thus the classifier is forced to learn to rule out zones in which it might think that it has detected parts of vehicles, and even filter the regions where vehicles are very close to each other, which is beneficial for accurate localization of vehicles. We also point out that as opposed collecting random negative training samples from different sources or internet databases this method collects samples specific to the scene we are observing thus it further reduce the risks of having false detections from the classifier. Another idea that we propose it to use the background, model obtained in previous stages, by segmenting it in random selected patches which we will also use as negative training samples. The classifier will learn scene specific features that it will easily distinguish as not containing vehicles, and further reduce the number of false detections.**

**We show in figure 4.9 and figure 4.10 the positive and respectively the negative samples that we end up being saved in the training set. Experiments have demonstrated that 100% of the captured positive samples are indeed containing a single, well centered vehicle. We also have to mention that we recorded having centered vehicles in negative samples also, mainly due to the fact that separate vehicle blobs were captured moving close and in the process of saving one of them, the other was saved as a negative sample. For increasing the accuracy of the classifier one might further go over all the negative samples and remove them manually, but one target of this project is to automate the process for acquiring the training set. Thus we did not hand-remove the negative samples that were containing centered vehicles and left the training set exactly as it was constructed in the described process of acquisition.**

We can observe that for our scene, the vehicles captured in our positive training set have little variations around a median orientation imposed by the road topology and also little variations around a median scale imposed by the camera placement. This is the effect of having a high placed video acquisition device surveilancing a relative linear topology of the road. This training set can offer a great depth of discriminative information regarding shape and locality concerning the vehicle class in contrast to other object classes that might pass through the scene.



**Figure 4.9 Extract of the positive samples training set**



**Figure 4.10 Extract of negative samples training set**

## Feature Extraction and Classifier

#### **4.2.1. Feature Extraction**

**One of the main challenges in developing a machine learning application concerns the features you use in training the classifier. This decision was taken only after we carried out experiments with different feature extraction methods and studying the results and improvements proposed in other papers.**

It is unavoidable that moving vehicles in the scene will suffer different scale and pose transformations depending perspective projection, their motion direction, elevation of the camera: their image may be as small as just a few pixels, or in close proximity they may be covering a large portion of the screen. The interactions and events that take place in an outside-door environment are of such high complexity that they cannot be properly understood by the computers in order to convey the semantic meaning.

Often vehicles will often be occluded: a vehicle passing behind a tree, a man passing in front of the car, or vehicles occluding each other. In the domain of tracking occlusion is a very complex problem which has been approached in many paper subjects. The reason for this is that is very hard to identify if an object is occluded by the background and other objects or it is occluding itself with new untracked parts.

As every classification problem the goal is to maximize inter-class variance or minimize intra-class variance. We want to be able to correctly recognize all type of cars, regardless of size, position, scale specific for our scene and also discard any kind of objects falling in other classes like pedestrians.

Our goal is to capture the variations in the data: different views of vehicles, different lightings, weather conditions, shadows, and so on.

Detecting cars is a considerably more difficult problem than detecting textured objects which have simple, semi-rigid structure, where the localization of components does not vary much between samples. Cars have a semi-rigid structure as well, but that structure will vary more between samples, because their shapes and configurations have been designed with product differentiation in mind. Besides the intra-class variations due to color, shape, and ornamentation, which similarly plague face and pedestrian detection systems, there are other issues that complicate car detection. Compared to the other object classes, vehicles also lack texture and have a highly reflective surface. Experimentally we observe, from the background subtraction technique, that the majority of vehicles tend to resemble the color of the pavement in some regions.

In the yearly days of our project we have analyzed the possibility of using the Bag of Words method for extracting a data vector that describes the image sample. In short this method extracts SIFT key points/features from the collection of training samples and clusters them to obtain cluster centroids which denote the most relevant features from the whole set. Next an SVM is trained using as feature vectors the frequencies of the most representative key points in each of the training samples. The disadvantage of this method is that it lacks locality, the position of the extracted key points is discarded, and also we were not obtaining the results that we needed.

At this stage of the process we were inspired by the works of [11] with the formulization of HOG (Histogram of Oriented Gradient) in the context of human detection and [6] with the application in the context of tracking vehicles on a highway surveillance.

The idea of Histogram of Oriented Gradient descriptors is to represent an image patch by the distribution of gradients directions. A good approach in describing a whole image is to divide in it a grid manner where each segmented region is called a cell and then computing for each a histogram of gradient orientations. The combination of these histograms in a single data vector will represent the feature vector or descriptor of an image. To increase accuracy and account for local illumination, shadowing, foreground-background contrasts an extra step can be performed, the histograms within a cell are contrast-normalized by computing a measure of intensity across a neighborhood of cells called a block, and this value is used to normalize cells within a block. The original description of this method along with schemes of how to customize the block normalization procedure can be found in [11].

After obtaining the training set the HOG descriptors are extracted off each training sample and saved as positive or negative descriptors which are later used in the SVM training phase. In our solution we divide each training sample into cells of size 8x8 pixels and the blocks are of size 16x16 pixels.

#### **4.2.2. Classifier**

The next step after obtaining the descriptors of the positive and negative training samples is choosing a classifier. For our work we decided to use SVM linear because they offer fast evaluation and fast training with the expense of choosing good training features.

## 4.3 Detector

The detector has the responsibility of applying the classifier over the frame in a sliding window approach providing robust detection region for the vehicles. As you have seen in figure 4.9 the vehicles that appeared in the positive training set have very close scales, due to the fact that they have been saved within roughly the same camera distance. The truth is that vehicles will appear at scales which are different from the one deducted from the training image set, therefore the detector has search at different scales of the image. It does this by iteratively decreasing the image size and keeping the size of the search window over the candidate image trying to identify vehicles that appear at larger scales. In our case we iteratively decrease the image by a factor of 1.05. Once the image has been scaled down several times the HOG histograms are computed for each of them. When sliding the searching window over the image it is important to have a stride in its displacement because moving it pixel by pixel is not efficient and does not even increase accuracy. In our solution we apply a stride equal to the cell size. At each position of the sliding window the histogram of the blocks is under the sliding window is extracted and feed it to the classifier. The classifier lastly compares the distance of the input histogram to the separating hyperplane learnt after termination of the learning phase and the detector records this as a response for the current position of the sliding window and continues to search new locations of vehicles by iteratively displacing the window top-down from right to left.

After the process of searching for vehicles in all scaled images we obtain a list of possible car locations each bearing a score received by the classifier. At this stage it is likely that at the real location of a vehicle we could have obtained multiple positive responses from the classifier. These positive responses are generated at small deviations of the sliding window around the true position and scale of the vehicle. The next task is to filter all these satellite detections and obtain the best one for tracking. We make this decision based on the classifier output that characterizes each location. This challenge reduces to the problem of finding disjoint sets. Specifically we apply to solution of disjoint sets in the goal of eliminating near close detections located under a specified threshold. After obtaining the disjoint sets, a voting procedure is carried out for each set to construct the position and size of best detection for that set, a representative, and finally if there are overlapped, detection representatives, above a selected threshold, then only one of them is saved depending on the size of the disjoint set that it has constructed it. Figure 4.11 show the strong responses that the classifier gave for the scene and figure 4.12 depicts the results of filtering the detections for a vehicle.

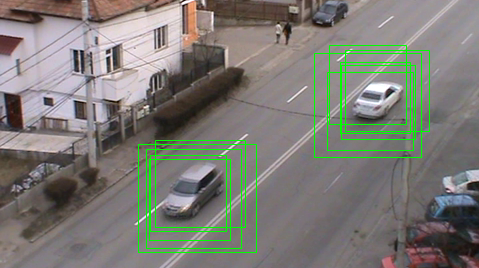


Figure 4.11 Multiple detections for one vehicle

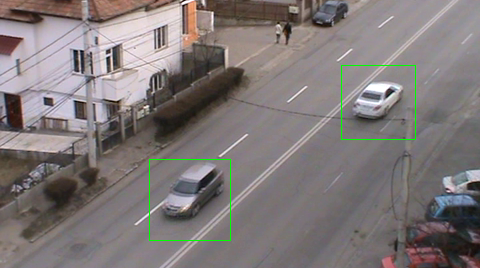


Figure 4.12 Filtered Detections

## 4.4 Background Subtractor

We have already discussed the Background Subtraction technique and used it in the process or forming our Training Set. The technique that we have used is called MOG (Mixture of Gaussian) and the principal strength of this approach in constructing the background model is the tolerance to noise but has the weakness that it cannot achieve the real-time standards that we are aiming to achieve. We have also mentioned that it is not a concern in the case of the Training Set Acquisition process.

This Background Subtractor module has the responsibility of providing the tracker with pixel level information of how is the scene segmented into static and moving objects, i.e. the background and potential moving cars. The tracker will further use this information to extract patterns from the surface of vehicles which can be submitted for tracking.

The main concern that we have to face here is the real-time performance. While this is not an issue in the case of the training set acquisition process, we have determined experimentally that that the quality of the foreground mask need not be high and thus we allowed ourselves to use a faster and simpler method for this stage in order to achieve a good throughput of foreground masks for the tracker.

The method that we have used here is based on maintaining a *running average* as the background model for each pixel, thus a pixel is represented by a single value, as opposed to a mixture of Gaussian distributions in the MOG. In [12] they have experimented with different subtraction methods and have concluded that the Running Average is certainly the fastest, in which the mask is just a thresholded difference between the background model and the frame, while the model adapts one or two parameters. This method is first mentioned in [13] and they propose a background model independently at each pixel location. The model is based on ideally fitting a Gaussian probability density function on the last *n* pixel’s values.

In order to avoid fitting the probability density function from scratch at each new frame time *t*, a running average is computed cumulative on line instead as:

where , is the pixel’s current value and , , the previous average, *a* is an empirical weight often chosen as a trade-off between stability and quick update, you can think of it as the learning rate. In addition to speed, the advantage of the running average is given by the low memory requirement for each pixel. At each frame time a pixel's value can then be classified as a foreground pixel if the inequality:

where *k* is a threshold above which the deviation of a pixel from the mean value leads to it being marked as an object pixel. Figure 4.13 is an example of a frame, the background model constructed until this frame and its foreground mask using the Running Average.



Figure 4.13 a. Current frame in the video



Figure 4.13 b. Background model



Figure 4.13 c. Foreground mask with *a=0.01 and k=20*

Figure 4.13 Illustration of the Running Average method on our video

As you can see from figure 4.13 the foreground mask is not of high quality and is the subject of noise. We have several options of improving the results:

* Increase the threshold *k*, this will increase the value with which pixels can vary around their running average thus reducing noise, but as a consequence cars with colors that closely resemble the pavement will not be detected. This effect is illustrated in figure 4.14
* Increase the learning rate *a*, this will lead to stabilizing the color of a pixel in the background model more quickly, it will converge to the current color faster, but as a consequence pixels covering large moving objects of constant color will be detected as background because the mean color of the pixel has stabilized too quickly when the object was in the phase of entering the pixel; a similar effect can be seen in the center vehicle of figure 4.13 where the bonnet and windshield are detected as background pixels.
* Apply morphological transforms, erosion and dilation, to filter the noisy regions, small connected regions surrounded by background; these are regions that mostly are generated by noise in video acquisition and movements of small objects which we are not interested in. Depending on the structuring element, with erosion we can filter a large portion of the noisy pixels in the foreground mask in figure 4.13, and with dilation we can close the holes in the blobs containing vehicles. Experimentally we have obtained satisfactory results by applying erosion followed by a closing operation. The structuring element that we have used is a cross of size 7x7 with the origin in the center of the cross. Figure 4.15 exemplifies the results.



Figure 4.14 Foreground mask obtained with an increased *k,*

*having a=0.01, k = 50*



Figure 4.15 Foreground mask by applying erosion

followed by a closing operation

## 4.3 Tracker

The tracking module is the most complex component of our solution and the subject of contribution approached and proposed by this paper. The aim of an object tracker is to generate the trajectory of an object over time by locating its position in every frame of the video. Object tracker may also provide the complete region in the image that is occupied by the object at every time instant.

Essentially our tracking module is responsible for managing the tracks of detected vehicles by:

* Initiating new tracks when new vehicles are detected
* Merging detections to tracks when detections are available
* Producing track estimations when detections are not available
* Scoring the tracks
* Identifying false positive tracks by the scoring method
* Counting true tracks generated by vehicles

We have discussed that in order to perform these goals the Tracker receives several inputs, detections from the Detector under the form of bounding rectangles and also foreground masks from the Background Subtractor. The Tracker combines these two sources of information by overlapping the detections over the foreground mask and thus obtaining the pixels from the foreground mask that are thought to belong to a single vehicle. This information is useful because we later use a point tracker.

The method proposed by this paper in the subject of tracking using two different techniques at its core, the Kalman filter and Lucas-Kanade point tracker. Each technique has its strong points and strengths. The main idea is to realize when one technique is not suitable to apply because specific situations exploit its weak points and use the other’s technique strong points to compensate. In what follows we will review the main ideas and assumptions that the Lucas-Kanade point tracker and the Kalman filter assume, what is the best case and worst case when applying these methods, what are their advantages and disadvantages. Based on these, we propose a technique, suitable for short term tracking, in which we dynamically decide which estimation is best for the initialized tracks by consulting the features that we know to belong to each vehicle within a track. Figure 4.16 is a sketch of the data flow of the tracking algorithm performed for managing each track that it has detected in the process. The tracker maintains a list of tracks from the previous frame. Each track is selected by the tracker and its position and size is estimated in the current frame. We perform two estimations, one with a Kalman filter and the other with a Median Flow filter which will be detailed later. In the step of merging the predictions, we evaluate which of them is the best estimation based on the features specific to the track. Next we try to match detections with track estimations, by verifying if the detection lies on the track trajectory and caries similar features. If such a detection is found then we merge it with the matching track by resetting the track features to the detection features. Additionally we must carry a scoring procedure for each track to determine if it is a false positive or true positive.

### 4.3.1 Kalman-Filter Estimation

At each frame we could have a detection of a vehicle and find ourselves with an estimate of its true location, which was provided for us by the Detector. This estimation is not likely to be extremely accurate. The reasons for this are many. They may include inaccuracies and noise in the video sensor, approximations in earlier processing stages, issues arising from occlusion or shadows, or the changing of shape. Whatever the source, we expect that these measurements will vary, perhaps somewhat randomly, about the actual values. We can think of all these inaccuracies, taken together, as simply adding noise to our tracking process. We would like to have the capability of estimating the motion of vehicles in a way that makes maximal use of the measurements we have made, specifically the detections. Thus, the cumulative effect of our many measurements could allow us to detect the part of the vehicle’s observed trajectory that is not subject to noise, inaccuracies or deviations of the detection around the true location of the vehicle.



Figure 4.16, Flow of the tracking algorithm

The key additional ingredient is a *model* for vehicle motion. In our case we model the vehicles motion under the following assumption: *A vehicle enters the frame at one side and passes through the frame with approximate constant velocity*.

Given this model, we can determine where the vehicle is. This task is divided into two phases. In the first phase, typically called the *prediction phase*, we use information learned in the past to further refine our model for what the next location of the vehicle will be. In the second phase, the *correction phase*, we make a measurement and then confront that measurement with the predictions based on our previous measurements, in our case the measurement is the detection.

The original formulation can be found in [15]. The Kalman filter builds a model for the state of the system that maximizes the a posteriori probability of those previous measurements. It means that the new model constructed after making a measurement is the model that has the highest probability of being correct, considering the previous model with its uncertainty and the new measurement with its uncertainty. In addition it is not required to keep a long history of measurements, i.e. detections, but iteratively update our model of a system’s state and keep only that model for the next iteration.

There are three important assumptions required in the theoretical construction of the Kalman filter:

* the system being modeled is linear
* the noise that measurements are subject to is “white”
* this noise is also Gaussian in nature

The first assumption means that the state of the system at time *k* can be modeled as some matrix multiplied by the state at time *k–1*. The additional assumptions that the noise is both white and Gaussian means that the noise is not correlated in time and that its amplitude can be accurately modeled using only an average and a covariance. The idea of the Kalman filter is to start with what we know, we obtain new information, and then we decide to change what we know based on how certain we are about the old and new information using a weighted combination of the old and the new.

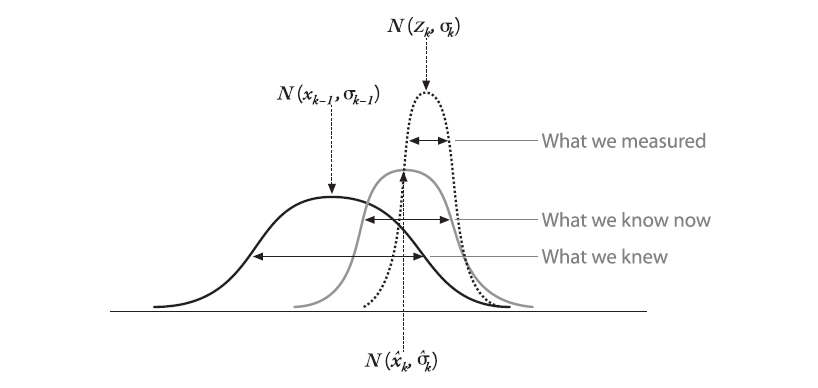


Figure 4.16 Source [14], Combining our prior knowledge with our measurement observation; the result is our new estimate

It is beyond of the scope of this paper to deduce the equations of the Kalman filter but they are briefly presented in figure 4.17 for each phase.

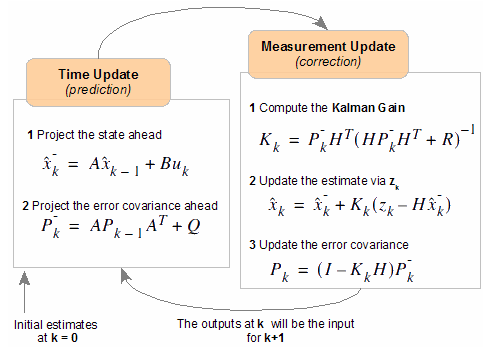


Figure 4.17 Source: [10], Kalman filter phase equations

where *A* is the transfer matrix, used to express linearly as a function of ; is the prior estimate before update correction; is the estimate after the measurement update ; is the estimate after the measurement of the previous phase; is denoting the external controls on the system, is used to relate these external controls, is the measurement we are incorporating in our model, in our case it is the position of the detections; is a matrix relating the state to the measurement; is the prior error covariance; is the Kalman gain which tells us how to weight new information against what we think we already know.

As mentioned we used the Kalman filter to estimate the position of the vehicles under the assumption of constant velocity, and the matrices of the filter that we used are:

, , ,

The measurement in our system represents the coordinates of the center of the bounding rectangle detection.

### 4.3.2 Lucas Kanade Optical Flow, Median Flow Filter

A good introduction on the Lucas-Kanade algorithm can be found in [14] and the original formulation can be found in [16]. The algorithm is a widely used differential method for optical flow estimation. The optical flow is a velocity field in the image which transforms one image into the next image in a sequence.

The LK (Lucas-Kanade) algorithm can be applied in a sparse context because it relies only on local information that is derived from some small window surrounding each of the points of interest. The disadvantage of using small local windows in Lucas-Kanade is that large motions can move points outside the range of the local window and thus they become impossible for the algorithm to find in the specified range.

The basic idea of LK algorithm rests on three assumptions, as highlighted in [14]:

* *Brightness constancy*. Image measurements, brightness, in a small patch remain the same although the location may change. This means that we can assume that pixel brightness does not change from frame to frame as we are tracking the point.
* *Temporal persistence or small movements*. The image motion of a surface patch changes slowly in time. In practice, this means the temporal increments are fast enough relative to the scale of motion in the image that the object does not move much from frame to frame.
* *Spatial coherence*. Neighboring points in a scene typically belong to the same surface and by projection onto the image plane we would expect the same special coherence in the image as well.

We will not go into details of deriving the equations but based on these assumptions we must highlight some worst case scenarios that our scene can exhibit to induce errors in this method of tracking:

* As we have already mentioned some parts of vehicles appear as high reflective surfaces thus have a high probability of changing their surface brightness as they pass through the environment.
* On the other hand, other parts of vehicles appear as patches of constant color thus the movement of neighboring pixels cannot be detected.
* Another phenomenon that takes place to break the third constraint is occlusion in which case points, placed the margin of overlapping vehicles, exhibit neighbors that have different flows. Such cases appear when vehicles produce occlusion among them, or when they are passing behind a pole for example.

This method work best in controlled environments, but in outdoor environments it is difficult to rely on tracking successfully points through a large number of frames. Due to the assumptions stated above the track generally suffers drifts for the actual path or can even be tracked in a completely wrong region.

To overcome the disadvantages of tracking the same points over a long number of frames, our approach is to only track newly selected points from frame to frame. The advantage over tracking the same points is that consecutive frames do not vary dramatically and so we can rely on the assumptions stated, while points visible in the first frames will often get occluded as they are carried by the car surface into the frames in which the car is exiting.

The method of how we use the Lucas-Kanade tracker to estimate the position and scale of a vehicle in the next frame is the one proposed in the works of [7]. They propose a method of self-evaluating the tracking reliability and estimating the displacement from frame to frame of a bounding box. The proposed method is based on the principle of forward-backward consistency of correctly determined tracks, more precisely, given a sequence of frames the track of an object should be independent of the direction of the time-flow. Algorithmically the method can be implemented as follows.

* First we track a point *forward*, over several time frames, obtaining the forward trajectory.
* Secondly we track the point from the time-frame we stopped to the time-frame we started; this is equivalent to reversing the flow of time as we pass through frames, thus producing the *backward* trajectory.
* Thirdly, we can compare these two trajectories and verify how consistent they are with one another. If the difference is significantly, then this means that it is not reliable to follow the forward trajectory. A simple scheme to evaluate the difference in trajectories which we also use is to compute the Euclidean distance between the initial location of the point and the location of the backward tracked point.

.

Figure 4.17 is an example of tracking a point from frame consecutively *k* frames, until frame . The points to construct the forward trajectory. Similarly the points denoted by , and so on make the backward trajectory from frame to frame . We are expecting that, based on the independence on the direction of the time-flow of the track, the re-tracked point would lie within very close proximity to point . If it is not the case then it can be concluded that point cannot be tracked with confidence and is marked as an outlier for further processing. A simple way to estimate the error in tracking is to apply a distance metric between and on the original frame; we are using a simple Euclidean distance.

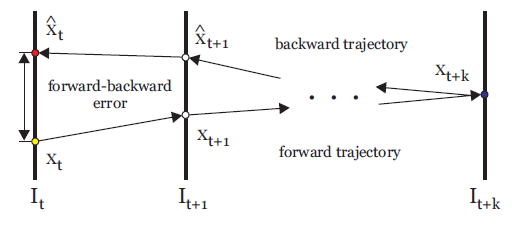


Figure 4.17 Source: [7], Forward-Backward trajectories and error evaluation

The authors of [7] also propose an object tracker called the Median Flow Filter. They formulize the problem of tracking an abject delimited by a bounding box. Their method is based on the observations that points are generally not independent, but are part of bigger objects or parts that move together. Figure 4.18 illustrates the algorithmic phases of estimation.



Figure 4.18 Source: [7], Median Flow Filter phases

The Median Flow filter accepts as input a bounding box around an object that we wish to track, within frame , and will output the updated bounding box in the next frame with the estimated location and scale of the object. The algorithm of the Median Flow is shown in figure 4.18 and has the following steps:

* Initialize a grid of points within the bounding box with a selected stride.
* Perform forward LK tracking with these points into the next frame.
* Perform backward tracking of the resulted points back into the original frame.
* Fix a threshold above which the distance between which the Euclidean distance between the initial points and the backward tracked points are considered outliers and can be discarded for the following computations because their tracking is not reliable. In our case this threshold is 2 pixels.
* Next estimate the displacement between frame and of the bounding box by computing the median of distances between the initial points and the forward tracked inlier points.
* For determining the scale we determine the ratio of distance between a pair of inliers in frame and the same pair in frame. The scale of the bounding box is estimated to be the mean of the computed ratios.

This method works best with rigid objects. Parts of the objects that do not satisfy this constraint because they exhibit a flexible nature and also object boundaries are discarded in the process of evaluating the inliers and are discarded. There is a possibility of keeping static inliers, which do not move from a frame to another, which are not part of the vehicle object. The displacement of these inliers will be zero and will affect the computation of the median displacement and also the median scale. To avoid this effect we have modified the algorithm to only select the points within the detections which are on the foreground mask, the blobs representing the vehicles.

### 4.3.3 Merging Track Predictions, Find Detection Match, Merging Detections to Tracks

At this stage of the algorithm we should decide how do the tracks model the identity of the objects (vehicles); we should ask which object representation is suitable for tracking vehicles? Which are the relevant features that uniquely identity an object: aspect, motion, shape? After answering this question we need to decide how should these features are abstracted in data structures and how do we extract them from the object. This decision relies on analyzing the environment and the goal for which tracking information is sought. Choosing features is critical in tracking, and in general it is preferable that feature representations of multiple objects these are easily separable in feature space. For example, color and contour representations usually used as features. In general, many tracking algorithms use a combination of these features.

After we have obtained two predictions upon the location of the vehicle for the current frame, from the Median Flow Tracker and the Kalman filter, it is necessary that we decide to assign one of them as the true location of the vehicle at this time step to the track history. Thus we want to determine what the best estimation that the trackers provided is. In order to evaluate this, we need to compare the features stored in the track for the vehicle it represents with the features that the estimations provided. The estimation which provides the best similarity measure with respect to the features of the track is considered to be the best, and the location and size of the track are updated to correspond with this prediction. Thus the *Merging Track Predictions* stage of the algorithm is determining the best prediction for which the distance between the features of the track and the prediction is minimized.

At each frame we are provided with detections on behalf of the Detector. These might be detections for vehicles which we are tracking, or detections for vehicles entering the scene. Either way we must always first resolve the detections with existing tracks, i.e. find to which track the vehicle in the detection is belonging. It is also very probable that for a number of frames we will not receive detections of cars which we have already started tracking. In this situation the tracks locations in the subsequent frames are estimated by merging the predictions from the Median Tracker and the Kalman filter. In the case in which we receive a detection for a vehicle which is tracked we need to update the vehicle features stored in this track. This way we are always updating the model of the car that the track stores to the last detection.

We are never updating the model of the track to estimation because estimations produce drifts from the actual tracks, only the Detector is responsible of recognizing the location and size of vehicles, thus the detections are used to correct the filters. In the case of the Kalman filter we are performing the measurement update, and in the case of the Median Flow we are resetting the bounding box from which it iteratively produces estimations.

Another reason for which we keep only the latest description of the detection in the track is that as vehicles passes through the scene they generally maintain their orientation and their scale typically increases or decreases gradually with a small factor from frame to frame, so it will be redundant to store previous feature descriptors as it improbable from the general movement on the highway that vehicles will reach a point in time that they would better resemble long distant extracted feature descriptors instead of the latest extracted feature descriptor.

We have discussed that we would need feature descriptors that uniquely identify the vehicles in order to robustly match vehicles belonging to initialized tracks with vehicles detections. Another important aspect to keep in mind is that the Detector will not accurately detect vehicles such that vehicles are perfectly centered in the detection bounding box, and moreover vehicles are constantly moving thus changing direction and scale. This tells that we should be careful in choosing the feature descriptors so that they are robust to small variations in orientation and scale. Next we present three feature descriptors that we have used.

### 4.3.4 Matching methods

#### 4.3.4.1 Location and Size Feature Descriptor

This is the simplest descriptor that one can use to model the detection and track. It simply describes the *x* and *y* coordinates along with the width and height of detections and tracks, thus the bounding box around the modeled vehicle. Although is appear not to have that discriminative power, we are using it in the process of finding the detection match for a track as a method to discard unlikely detection candidates that could not be merged with the track. The assumption on which we are applying this procedure is that only detections which are located on the trajectory of the track are suitable to be tested. This evaluation is done by computing if the intersection of the bounding rectangle of the detection and the bounding rectangle of the estimation of the track is above a sustain threshold, which in our case we compute it dynamically as being 60% of the smallest bounding rectangle area.

#### 4.3.4.2 Normalized Cross Correlation (NCC)

In the case of using the Normalize Cross Correlation, as a method to measure the similarity between the track and detection, we use the whole image under the detection bounding box and track as the feature descriptor, which we will refer to as the template and the image to be matched. In the case of image processing applications the NNC is suitable for situations in which the brightness of the image and the template varies due to lighting and exposure conditions. To account for this variation the images must be first normalized. This operation can be done online at every pixel location by subtracting the mean and dividing by the standard deviation. The cross *correlation* between the image *f* and the template *t* is in the point (*m, n*) is:

here is the subimage mean with the center in (*m, n*); is the mean of the template.

In our case we want to examine the similarity between entire images determined by the detection and track and we are expecting that these two are not of equal size. For this reason we downscale the biggest image to the size of the smallest and apply cross correlation. It is also a good idea to blur the images before applying NCC.

Experiments have shown that NCC is not a good measure of similarity for our application, and matching is poorly when detections do not center the vehicle accordingly to the model that was saved in the track. These deviations of the detections around the true location of the vehicle produce high scores when applying NNC between the track and incoming detections from the Detector.

#### 4.3.4.3 Local Bit Patterns (LBP)

Local Bit Pattern is an operator applied on textures which determines the label of a pixel by thresholding its neighboring pixels. The idea of comparing two textures based on local bit patterns relies on local spatial patterns and gray scale contrast.

The basic idea is to summarize the local structure in an image by comparing each pixel with its neighborhood, i.e. we take a pixel as center and threshold its neighbors against. If the intensity of the center pixel is greater-equal its neighbor, then denote it with 1 and 0 if not. You'll end up with a binary number for each pixel, just like *11001111*. With 8 surrounding pixels you'll end up with  possible combinations, which are called sometimes *LBP codes*. The first LBP operator actually used a fixed *3 x 3* neighborhood, but the neighborhood can be extended to a circular pattern, which we are using, refer to figure 4.20 for results.

As formulated in [17] the LBP can be used in texture classification by collecting the resulted labels, after applying the operator, into a histogram. The classification operation then relies on a measure of comparing histograms. However by relying only on the histogram to serve as a feature descriptor we arrive at loosing important spatial information, i.e. the location from which the labels were collected is important.

One way to overcome this disadvantage is to compute the LBP descriptors on local patches and then concatenate them in a global descriptor. Such descriptors are widely used because the offer robustness against small variants in pose and illumination.

This method of using the LBP descriptor was used in [18] within the scope of face recognition. The steps that they performed in obtaining the global descriptor are as follows, also depicted in figure 4.19:

* The image is divided into local regions
* LBP is used to extract the histograms from each region
* The histograms are concatenated in a global descriptor.

The histogram effectively operates on three levels of locality:

* Labels from which the histogram is collected contain information on a pixel level
* Summarizing the labels over small regions produce local patch information
* By concatenating the histograms, resulted from different patches, we obtain a global descriptor which keeps the information about patch locality.

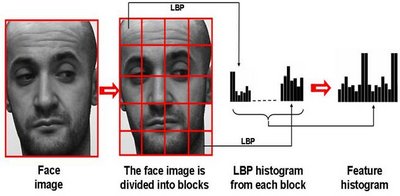


Figure 4.19 Source: [20], Forming the LBP global feature

Description: E:\Academic\Licence\lic_doc\pics\chapter 4\lbp-279.png Description: E:\Academic\Licence\lic_doc\pics\chapter 4\lbp-54.png Description: E:\Academic\Licence\lic_doc\pics\chapter 4\lbp-235.png Description: E:\Academic\Licence\lic_doc\pics\chapter 4\lbp-8.pngDescription: E:\Academic\Licence\lic_doc\pics\chapter 4\lbp-159.png

Figure 4.20 Circular LBP results

### 4.3.5 Scoring tracks

In our experiments a very simple scoring function is employed which is inspired from the works of [19]. Each track starts with a score of 0. Then for every detection merge in the track, the score is increased by a value determined by the number of consecutive detections up to this one. We perform the same step if we have an estimation whose similarity with the track is below a selected threshold .If we do not have a detection or an estimation below the threshold of similarity we decrease the score by a constant value, 1 in our case. Finally, only the tracks with positive scores and above a specific threshold are accounted as belonging to vehicles. This last threshold is selected by observation and we selected it such that vehicles which are tracked from the middle of the frame to be scored as positive. This method of scoring the tracks has the advantage of boosting the scores of the tracks which had consecutive or continuous detections.

# 

# 5. Detailed Design and Implementation

In this chapter we will present the reader with all the information necessary to understand the work done on the application, so it can be extended and improved in a future development. We will start by describing the reason behind the chosen language, tools, the top architecture, and then we will talk about the design of the application going into the most important details of the implementation. We will try to highlight the packages that our application in structured into, try to describe what functionality do packages organize and try to reconstruct the structure with class diagrams.

After this overall view of the entire project, we will start to explain the actual implementation in terms of source code. We will then try to present the implementation in a straight-forward way, such that anyone with basic understanding of the programming language in which it was written (C++, and small Ruby scripts) should be able to take a look over it and understand and maybe extend its functionality.

This section will also contain the code of the most important methods used along with their explanation.

## Language, Technology, Frameworks

#### 5.1.1 C++ OX-11

Our tool was created using C++ programming language using the new standard adopted last year in August. Areas of the core language that were significantly improved over the last version include multithreading support, generic programming support, uniform initialization, and performance enhancements.

One should take a look at the improvements and new standards adopted for this version, and which the Visual Studio 2010 compiler included before their standardization, because we will use some of them, and these features are:

* New struct initializers
* Lambda expressions
* Type inference
* Foreach iterations over containers
* Smart pointers(unique\_ptr, shared\_ptr) which act as reference counters and which make memory management similar to garbage collected based languages

#### 5.1.2 OpenCV

As pointed out in chapter 2 this is the main framework that we rely on to do the heavy lifting in terms of performing well established algorithms. We will rely on constructs that provide processing in the domain of:

* Background subtraction
* Classification
* Detection
* Lucas-Kanade Optical Flow
* Kalman Filter
* Morphological transforms

#### 5.1.3 Asynchronous Agents Library and Parallel Patterns Library

According to [21] the Asynchronous Agents Library (or just *Agents Library*) provides a programming model that lets you increase the robustness of concurrency-enabled application development. The Agents Library is a C++ template library that promotes an actor-based programming model and in-process message passing for coarse-grained dataflow and pipelining tasks. The Agents Library builds on the scheduling and resource management components of the Concurrency Runtime. The AAL and PPL provide the following features:

* *Task Parallelism*: a mechanism to execute several work items (tasks) in parallel
* *Parallel algorithms*: generic algorithms that act on collections of data in parallel
* *Parallel containers and objects*: generic container types that provide safe concurrent access to their elements

#### 5.1.4 SVMLight

This is a small executable downloaded from [22] offering implementation of Support Vector Machines (SVMs) in C.

#### 5.1.5 Ruby language

Ruby is a dynamic, reflective, general-purpose object-oriented scripting programming language. We will mainly use it in small scripts for file processing, particularly for interpreting the output from SVMLight to feed it back to the OpenCV Hog classifier and detector.

## 5.2 System packages

In this section we will try to show the current design and implementation pointing out the modularity of the architecture and the contracts that it provides for the plugability of other modules that implement equivalent algorithms. The next figure illustrates the packages that compose the entire solution. The packages are built on top of one another by implementing the contracts that the lower level interfaces offer or by extending lower level classes. We have tried to follow SOLID principles in developing the overall architecture and designing each module. These principles when applied together intend to make it more likely that a programmer will create a system that is easy to maintain and extend over time.

The packages and their main concerns and responsibilities are listed below.

* VtInterfaces
  + Provides the interfaces and contracts that modules must implement in order to be plugged in the system
  + Provides necessary data structures of input and output for the interfaces, which implementations must use to transfer data
  + Also provides basic tools for basic geometrical computations
* VtImplementation
  + Provides interfaces implementations with the algorithms that we described
  + Also provides tools for processing member data structures that they auxiliary define
* TrainingSetAqusition
  + Provides an architecture that is used for processing the video to collect the dataset
  + Uses the interfaces from the VtInterfaces package to define the architecture and instantiates it with specific implementations from VtImplementation
* ClassifierTraining
  + Organizes the functionality to process the training set and extract training features
  + Performs classifier training
  + Collects output to feed it back into the system
* VehicleTracking
  + Provides the architecture of the tracking functionality
  + Integrates the modules of classification, detection and tracking in terms of interfaces
  + Offers the possibility of configurability with new algorithms by instantiating the interfaces to modules that conform to these interfaces

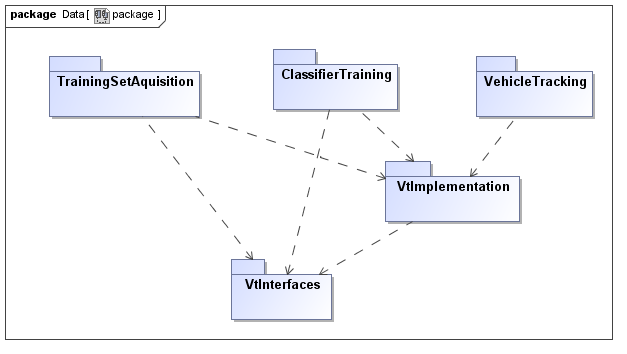


Figure 5.1 System package diagram

### 5.2.1 Vehicle Tracking Interfaces (VtInterfaces)

Figure 5.2 presents in detail the inner packages and classes that the VtInterfaces (Vehicle Tracking Interfaces) package organizes. This package provides the interfaces and contracts that higher level modules must implement in order for them to be plugged in the system. It also provides the necessary data structures of input and output for the operations of the interfaces, which implementing modules must use in order to transfer data. This package also has the responsibility of providing basic tools for drawing simple geometrical shapes, like points, circles, lines, rectangles and also text. Also, because we will be heavily process bounding rectangles, we also supply here classes for handling diverse operations on rectangular forms. The package also organizes two classes that handle file and other asset types loading, i.e. images, matrices, binary storage.

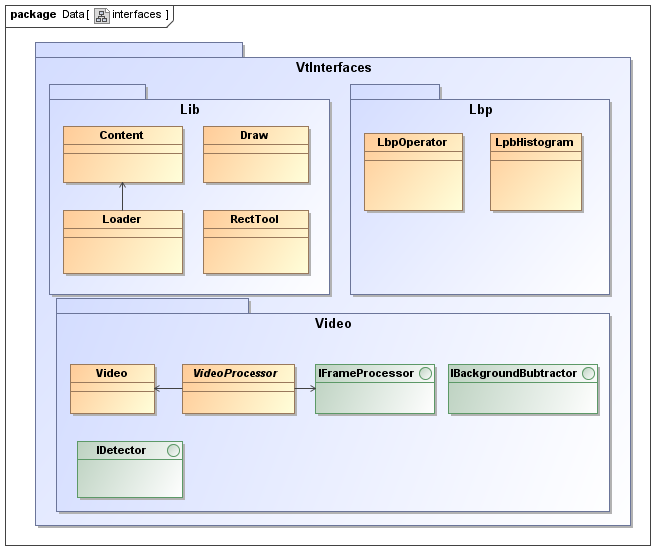


Figure 5.2 Inner packages and classes of the VtInterfaces package

As stated this package organizes the contracts of the application modules, by providing the interfaces and abstract classes that need to be implemented or overridden by concrete classes in order to be plugged in in the whole system. Also this is the package that we integrated third party open source implementations that are not exposed to change in the process of development. Next we will present what are the responsibilities and requirements that these components require from their implementers.

The **Video** class is responsible for opening video files and their properties, i.e. frames per second, frame delay, given path it in the constructor. The *readFrame(Mat& frame)-method* is capable of writing in the parameter passed by reference the next frame of the video at every call.

The **VideoProcessor** is the class responsible for opening a video and managing a list of **IFrameProcessor** interfaces which are subscribed to process every frame of the video. The responsibility of this class is to feed frames to implementers of IFrameProcessor. As you probably guessed, the **IFrameProcessor** interface is offering a contract on how to consume frames by the *process (const Mat& frame)-method,* this being the method that the VideoProcessor will push frames into.Once it has acquired a frame the implementers are free to perform a diverse set of algorithms on it, i.e. Canny Edge detection, blob detection etc.

The **IBackgroundSubtractor** interface is the one that must be implemented by the modules that perform background subtraction. This interface has the methods by which a implementing class of this interface is required to learn, segment, and return the model of the background it has constructed so far, i.e. *learn(const Mat& frame)-method*, *segment(const Mat& frame)-method*, *getBackgroundMethod()-method*.

**IDetector** is the interface that must be implemented by modules that must perform detection tasks (blob detection, vehicle detection). It can receive the frames on which it can perform its algorithm and output a data structure encapsulating a rectangle, you can think of it as the bounding box of the detection, and a label it has received in the detection process.

The **Content** class is static and has the responsibility of generating relative paths to the folders that we keep our assets, i.e. paths to files, images, xml files, videos.

The **Loader** class is also a static class that uses the Content class to provide dedicated methods for loading videos, images, xml files.

In the Lbp inner package we have two classes: the **LbpOperator** which performs the simple and circularly local bit panther operators on a input image and outputs the result. Next to that is the **LbpHistogram** class which is responsible for taking the result of a local bit pattern operation, segmenting this into blocks, extracting the histogram from each block and combining them into a single global histogram.

**RectTool** is another static class that provides many methods of working with bounding rectangle, such as moving, scaling, setting the width and height of rectangles, computing the area or computing the center.

The **Draw** class is also static and offers shorthands for calling the OpenCV methods for drawing points, circles and rectangles.

### 5.2.2 Vehicle Tracking Implementation (VtImplementation)

The VtImplementation (Vehicle Tracking Implementation) package contains inner packages and classes that offer the implementation of the interfaces from VtInterfaces packages of well-defined algorithms that are either callable from the OpenCV library or our customized implementations of them.

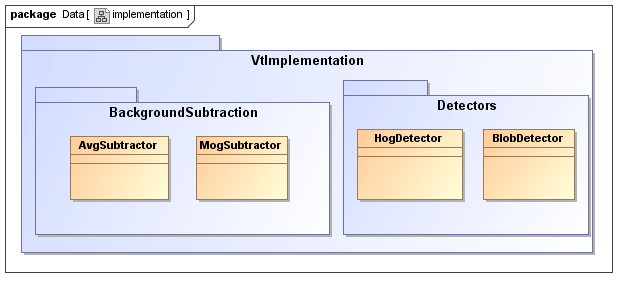


Figure 5.3 Inner packages and classes of VtImplementation package

As shown in figure 5.3 there are two main inner packages that group the algorithms covering the background subtraction operations and detection methods.

The **AvgSubtactor** class is implementing the Running Average method of background subtraction. As presented in chapter four it is responsible for constructing a background model of the scene by maintaining a running color average for each pixel on the image plane. The idea based on which we implemented this functionality is based on storing another image (Mat, which is the data structure from OpenCV for images and matrixes) in which we place running averages at the corresponding locations of pixels. The method that performs this operation is illustrated in figure 5.4. This background model is maintained in a floating point matrix. In order to compute the foreground we convert the floating point matrix into a gray scale image from which we subtract the gray frame. As last steps we update the background model as a weighted sum between the previous background model and the current gray frame, and apply erosions an dilations to reduce noise and close holes.

The **MogSubtactor** class is implementing the Mixture of Gaussian method of background subtraction. We are using the available class implementation from OpenCV to perform this algorithm, BackgroundSubtractorMOG2. We are contracted to implement the learning method of the interface whose implementation almost identical to updating method. Figure 5.5 is an extract from the MOG implementation.

In the Detectors package we find the implementations of the two methods of detecting objects, the simple BlobDetector, which acts at pixel level, and the HogDetector acting at semantical level of pixel groupings.

The **BlobDetector** is responsible for finding the connected components on a foreground mask by running an algorithm of contour following. Figure 5.6 shows the OpenCV calls necessary for finding these components. It is simple to see that for finding the contours we call *findCountours* after which we filter the contours based on their size and the locality in the detection frame.



Figure 5.4 Code for computing the Running Average background model



Figure 5.5 Code for computing the MOG background model

Figure 5.6 Code for detecting blobs

The **HogDetector** class is more complex because we are using the OpenCV implementation of a detector, the cv::gpu::HOGDescriptor. The name is very misleading and some developers have complained about it because this class combines the functionalities of a HOG detector, HOG descriptor and a Linear SVM which you must load with the supporting vectors in order to be able to perform detection. The main difficulty from using this class arises from the complete lack of documentation of using it, so you have to scrap different web sites to learn to use it. Another important aspect to remember is that you cannot use the SVM implementation from OpenCV to train it, and then load it into the HOGDescriptor because you need the supporting vectors as well as bias of the hyperplane. The tool which provides these for us is SVMLight. We will explain later how we use it and read its output. The code in figure 5.7 shows the necessary calls for applying the classifier over the frame in a sliding window approach, namely by calling *detectMultiscale-method.* We should also mention here that the current GPU implementations that this method will run on are CUDA capable NVidia GPUs, and you should have the appropriate OpenCV gpu library linked to your project.



Figure 5.7 Code for GPU HOG detection

### 5.2.3 Training Set Acquisition

As discussed in chapter four in the process of acquiring our dataset we must iterate through the frames of the video and process each one of them independently. As described we are not interested in the whole frame, but we will only process a region of interest reselected from each frame that the user selects to have the property that the vehicles passing through that region will appear to be minimally occluded. Since we have no information what are the features of the vehicle class at this point we must operate only with pixel information. But before just saving the images centered and containing these blobs we must apply some minimal tests be sure we are not saving other objects or more vehicles. Implementation wise we first create an instance of a VideoProcessor to which we subscribe an implementing instance of IFrameProcessor, namely an instance of the BlobExtraction class.

The BlobExtraction instance will receive every frame from the video thus is responsible for running the appropriate algorithms and managing the state of the application. This application has two states: the first state is responsible for capturing the input of the user, i.e. the bounding box the region of interest, and the second state is processing this selected region of interest from the rest of the frames. For the first state the BlobExtraction class stops the video at frame four, but it could be any starting frame, listening for the drag of the mouse on the screen to describe the region of interest. After this region was selected the user can proceed to the next state in which the extraction commences by pressing any key on the keyboard. At this step of processing we are not aware of any complex features of the vehicle class and thus we need to operate at pixel level, we will rely on the property of the region of interest stating that generally only vehicles are passing through that region and, moreover, they will appear in minimal occlusion with each other. We are interested here to obtain a high quality foreground mask from the scene in which vehicle blobs appear well separated from each other and not appearing to be linked due to noise or impossibility of the background model to fast adapt to these situations. We also mentioned that since we are in the training phase we do not care about the performance of the system, thus we will afford to run a time expensive algorithm which provides high quality results, Mixture of Gaussian.

The BlobExtraction class uses by encapsulation an interface of IBackgroundSubtractor which is instantiated to a MogSubtractor type. Lastly the result of retrieving the foreground mask is fed to an IDetector interface. This interface is instantiated to a BlobDetector type which is capable of finding the connected components from within the foreground mask. The found blobs are further discarded if the circularity of the convex hull is now within a threshold. Once the blobs are filtered a clipping rectangle of fixed sixe is placed over the center of the blobs then the frame is clipped and the clipped image is saved as a positive training image. For saving negative samples we place the same fixed sized clipping rectangles over the corners of the rectangle that has clipped the positive sample image. Figure 5.8 is the class diagram for this subsystem.

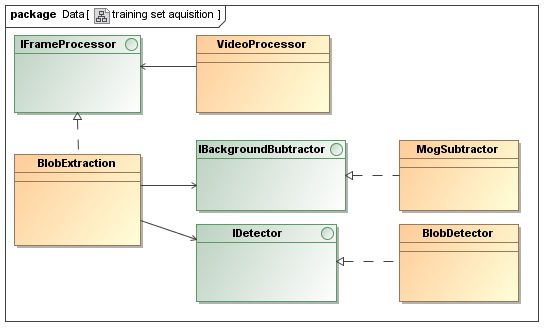


Figure 5.8 Training Set Acquisition class diagram

### 5.2.4 Classifier Training package

The ClassifierTraining package has the responsibility of processing the training samples collected to extract the necessary features for training the classifier, in our case HOG features. This is a simple executable that runs through the positive and negative samples extracting the hog features. The OpenCV class that performs the extraction is also the **cv::HOGDescriptor** with the *compute()-method.* Once the HOG features vectors are extracted we save it in a file (hogtrain.txt) that will be processed by the SVMLight library. After invoking the command:

.\Tools\svm\_learn.exe -z r -c 0.01 -i 0 -t 0 .\Assets\hogtrain.txt .\Assets\hogmodel.txt

the SVMLight executable will output the result of the training in the hogmodel.txt file, the file being populated with the support vectors, alpha values, and bias. In order to load the support vectors into the HOGDescriptor SVM we first need to process the results from hogmodel.txt to a matrix that is easily loadable in OpenCV, this matrix will contain the separating hyperplane that the HOGDescriptor SVM will need to know about. A simple Ruby script will make a yml format loadable matrix from this file. This single row matrix is obtained by multiplying the alpha values by their vector and summing the results of the multiplications together, which are also vectors. The last step is to append the bias to this summation, thus increasing the size of it by one. This is the matrix that needs to be set on the HOGDescriptor SVM.

### 5.2.5 Tracking

#### 5.2.5.1 Tracking subsystem design

This package contains the most important and complex subsystem of the application. It is responsible for integrating the described interfaces and their implementations to produce the input for the tracking algorithm. Aside from the class diagram we will need to discuss about the dataflow and the Asynchronous Agents Library because the architecture is centered about the high level features that this library provides to run code on threads and communication via synchronized containers.

Figure 5.9 is the class diagram of the tracking system. Before we go into details we should offer a few concepts on how the features that the Asynchronous Agents Library offers to multithread your code.

From the Microsoft guidelines in [21] we have leant to use the Agents library for multiple operations that must communicate with one another asynchronously. Message blocks and message-passing functions allow us to write parallel applications without requiring synchronization mechanisms such as locks thus letting us to focus on application logic. The agent programming model is often used to create data pipelines or networks. A data pipeline is a series of components, each of which performs a specific task that contributes to a larger goal. Every component in a dataflow pipeline performs work when it receives a message from another component. The result of that work is passed to other components in the pipeline or network. The components can use more fine-grained concurrency functionality from other libraries, for example, the Parallel Patterns Library (PPL).

Another concept that we use are the asynchronous message passing containers. The **Concurrency::unbounded\_buffer** class represents a general-purpose asynchronous messaging structure. This class stores a first in, first out (FIFO) queue of messages that can be written to by multiple sources or read from by multiple targets. When a target receives a message from an unbounded\_buffer object, that message is removed from the message buffer. The unbounded\_buffer class is useful when you want to pass multiple messages to another component, and that component must receive each message.

Figure 5.9 shows the classes that inherit the **agent** class from AAL and figure 5.10 depicts the data flow between the instances of these classes and the asynchronous communication buffers. All classes that inherit the agent class have to override the virtual method *run*() , where we have to implement the logic for each component.

The **PVideo** (Parallel Video) encapsulates a **Video** instance class, which is a wrapper over the OpenCV functionality of reading videos, and is responsible for reading the frames from the specified video path on a different thread. Next each read frame is cloned and sent along the unbound buffers that lead to the input of PSubtractor and PDetector instances.

The **PSubtractor** encapsulates an instance of instance of IBackgroundSubtractor, and delegates the background subtraction operation to the implementing instance once it has been notified that new frames are pending in its input unbound buffer. The implementer of the IBackgroundSubtractor interface is performing the Running Gaussian Average method for evaluating the background model. As we have already discussed, at this stage we are interested in maximum performance and the reason for which we are using the Running Gaussian Average is that it has low memory requirements, and the evaluation of each pixel is based on a simple subtraction and an absolute value comparison. While the Mixture of Gaussian method would yield better results, it comes at the cost of performance, i.e. the verification of every pixel relies on evaluating several Gaussian distributions, and thus the evaluation of the entire frame would produce great overhead.

The **PDetector** instance is also running on another thread, and each time it has been notified that a new frame has been committed on its input unbound buffer it consumes it from the buffer and delegates the detection operation to its encapsulated instance of the IDetector interface, which in this case is performing the detection based on HOG features, searching the frame at multiple scales in a sliding window manner. We wish to emphasize the operations of background subtraction and detection are performed in parallel, on two different threads.

If you will study other implementations of tracking systems you will probably find that the detection operation is performed after the background subtraction operation because it awaits the foreground mask. The detector would use the mask to determine where to apply the classifier. This is a good approach if you are using only the CPU for these operations. We have mentioned that our version of detector runs on the GPU. The reason for which we prefer to scan the whole image, without taking into account the foreground mask with the location of moving pixels, is that copying and changing textures on the GPU is an expensive operation and thus is faster to make a single call to the GPU with one large texture then to make more calls with small textures. The first implementation of our system was conforming to the idea to make the detections only on the foreground moving pixels and have concluded that performance was much lower than our current implementation.

The **PTracker** instance, which is also running on a separate thread, is waiting on its input from the foreground and detection buffers. Once these buffers both have a foreground mask and detection for the next frame, it consumes both these entries from the input unbound buffers and forwards the data structures to the tracking algorithm. The PTracker instance uses the estimator implementations, Kalman Filter and Median Flow Filter, for evaluating the location and size of tracked vehicles in the next frame, searches the detection that matches these tracks and lastly scores the tracks and outputs the annotated results.

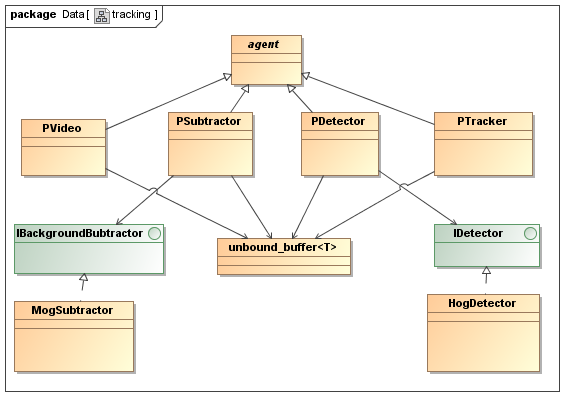


Figure 5.9 Class diagram of the tracking subsystem



Figure 5.10 Data flow diagram of tracking subsystem

#### 5.2.5.2 Tracking Estimators

As mentioned in chapter four the idea of the algorithm is to estimate the trajectory of the tracks, which we are aware of, in the current frame using two methods: The Median Tracker and the Kalman Filter techniques. Once we have these estimations about the position of the vehicle in the current frame we only consider the one which minimizes the distance to our current knowledge/model of the car. We have implemented three ways to model the features of cars and compute the distances between these features. After we have determined what the best estimation is, we try to match a track with one detection from the collection of detections arrives for this frame. Also at each frame we score the tracks in order to obtain the confidence of marking tracks as containing vehicles or false detections.

In our implementation we are using the Kalman filter predict the position of objects under the assumption of constant velocity. The OpenCV library provides an implementation of the Kalman Filter functionality wrapped in the class with the same name: **KalmanFilter**. We have developed the **KalmanFilter2D** class which encapsulates an instance of the type that we mentioned OpenCV has, and configured it with our system dynamics: transfer matrix, measurement matrix; and also provide methods which take application specific data structures (data structure containing x and y position of a vehicle – KalmanInput struct) which are then mapped to specific matrices of measurement to be to the Kalman filter for update. We have also provided a specific data structure which maps the matrix result into a simpler source of information. Figure 5.11 illustrates the interface of our class adapted for the case of tracking the position of objects.



Figure 5.11 KalmanFilter2D header

The **Median Flow Filter** is implemented in the PTracker class and makes heavy use of the Lucas-Kanade Point Tracker which is also exposed in the api of the OpenCV library and presented in figure 5.12.



Figure 5.12 OpenCV call to track points from one grayscale image into another

Its implementation is segmented across several function calls which initialize the tracking for the current frame, register tracks for estimation, performing the estimation for all tracks at once and reading the results back for each track. The signatures of these functions are presented in figure 5.13.



Figure 5.13 Signatures of the Median Flow tracking functions

The first implementation of the system was performing the median flow prediction separately for each track and this was a poor decision because it was necessary to perform point tracking from one frame to another for every track, instead of tracking all the points from all tracks once and them recovering them back for every track. Figure 5.13 shows the functions that you need to call to perform track estimations, all at once. The first function clears the state of the member variables involved in this procedure.

The second function registers a track for estimation by collection the points off of it and storing them. Once all the tracks have been registered you can call the *performTracking()-function* which tracks all the points from the registered tracks from the previous to the current frame. Once the tracking is finished you call the *getMedianFlowPrediction()-function* with each track that was registered outputting via the second parameter the estimated bounding rectangle, and the returning *bool* is an indication that the tracking for this track has been completed with success or not. The code for the tracking algorithm which integrates and coordinates the operations of the mentioned trackers and other supplementary functions is illustrated in figure 5.14. The section of code is dramatically simplified so that the main steps can be identified easily. In the while loop we first read the foreground and detections from the input unbounded buffers. The first region sets the variable needed for the current frame. The second region encapsulates code that takes care of the first detections by instantiating a track for every detection. The third region iterates through the existing tracks and performs Median Flow and Kalman estimation and then merges the predictions in the track. Once the positions and model of the track have been estimated for the current frame we iterate through the detections of the current frame to check which detection is identifying the vehicle on the track. If we found that we have a detection for a tracked car the we override the model of the track with information provided from the detection. Lastly we consider that all detections which were unmatched by tracks are considered to be new vehicles. In case we get further detection of these new vehicles then its score will increase, but in case the first detection was a false positive then the score will decrease and be removed from the tracks.



Figure 5.14 Simplified tracking code

# 6. Testing and Validation

## 6.1 SVM Training Error – Precision and Recall

As explained in [23], in classification tasks, the terms *true positives*, *true negatives*, *false positives*, and *false negatives* compare the results of the classifier under test with the actual ground truth. The terms *positive* and *negative* refer to the classifier's prediction (sometimes known as the *observation*), and the terms *true* and *false* refer to whether that prediction corresponds is conforming to the ground truth (sometimes known as the *expectation*). This is illustrated by the table 6.1.

|  |  |  |
| --- | --- | --- |
|  | actual class (expectation) | |
| predicted class (observation) | tp (true positive) | fp (false positive) |
| fn (false negative) | tn (true negative) |

Table 6.1 Ground truth – classifier results comparison

Precision and recall are then defined as:

,

The video which we worked with was 40 minutes long, on which we have performed different runs of the training set acquisition process within the first 15 minutes, varying the region of interest thus acquiring training images of different poses and size. Within these 15 minutes we managed to store approximately 1000 positive samples and 4 times as many negative examples. For our classifier training we have varied the number of positive and negative examples around 1000 or less for each different runs. Table 6.2 shows the svm training properties and results for the best svm which we obtained and working with.

|  |  |
| --- | --- |
| Positive Training Samples | 999 |
| Negative Training Samples | 999 |
| Support Vectors | 863 |
| Precision | 98.51% |
| Recall | 99.20% |
| Accuracy | 98.85% (1975 correct, 23 incorrect, 1998 total) |

Table 6.2 SVM training results

Next we tested the classifier on two test sets, which were composed by images that were not part of the SVM training process and which were extracted within a different data set extraction process so that the test set could exhibit different poses and size then the training set. Thus we had to spare around 400 positive samples and have chosen to test 1000 of our available negative samples. The accuracy on the positive test set was 94.74% - 378 samples were correctly classified while only 21 were incorrectly classified from a total of 399 positive test samples, thus the precision/recall on this test set was 100% / 94.74%. Also, out of the 1000 negative test samples 973 were correctly classifier while 27 were misclassified resulting in 97.30% accuracy and the precision/recall was 100% / 97.30%. We were expecting that the result to be this good because our observed scene is simple and observed from a high viewing point and although we tested the classifier on samples extracted from another region of interest, these samples were not expositing that much variation in size and pose to other extractions.

## LBP-NCC Comparison

#### 6.2.1 Tracker Accuracy with LBP

In such an application is hard to evaluate the tracker in an automatic way as we do not have the ground truth to compare results with for our video. Thus the only method that remains in evaluating the tracker is by human observation by evaluating in the save time the ground truth and the output that the tracker is giving. In the next figure we are showing an approximate comparison of the ground with the tracker results at different time steps in the video. On the horizontal axis is the time flow, and the vertical axis we record the number of vehicles that have passed though the frames as the ground truth and also the tracker estimation for the vehicle count, for comparison. We hope to establish if the tracker underestimates or overestimates the true number of passing vehicles.

From the curves in figure 6.1 we notice that the tracker does not over score the tracks, and its was to be expected that vehicles exhibiting unusual positions and collision will not have enough detections in order for them to be properly scored. Although low detected vehicles will initiate new tracks the lack of detections will cause them to be negatively scored and removed, or on exiting the frame their score will not be high enough to trigger a vehicle recognition action from the tracker.

#### 6.2.2 Tracker Accuracy with NCC

The NCC matching suffers from same problems as the LBP tracker. The NCC method of deciding if a detection can be merged with a track is too restrictive. The NCC matching method is directly related to the robustness of the Detector. Normally the Detector will not provide detections in which vehicles are perfectly centered in the detection bounding rectangle, but will vary around the true position of the vehicle. As it should have been expected, while NCC produces good results, it is not robust to pose variation and variations in detections relative locality to the actual vehicle. A comparison of the vehicle count maintained by the NNC based matcher with the ground truth is illustrated in figure 6.2.

Figure 6.1 Ground truth – LPB Tracker comparison

Figure 6.2 Ground Truth – NCC Tracker comparison

In table 6.3 we summarize the performance of the two tracking methods. The GT row stands for the ground truth vehicle count recorded at different time instances. The LBP row contains the number of vehicles that the LBP Tracker has recorded until that time instance. Similarly the NCC row contains at each column position the number of vehicles that it has scored as positive. We can observe that local bit patterns perform a better job than the normalized cross correlation, due to its tolerance to local region similarity.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| GT | 15 | 21 | 45 | 63 | 100 | 178 | 250 | 300 | 348 |
| LBP | 13 | 19 | 42 | 58 | 94 | 167 | 238 | 284 | 326 |
| NCC | 13 | 20 | 41 | 55 | 90 | 157 | 221 | 265 | 307 |
| Error  LBP | 1.3% | 9% | 6% | 7% | 6% | 6.1% | 5.3% | 5.3% | 6.3% |
| Error  NCC | 1.3% | 4% | 8% | 12% | 10% | 11% | 11% | 11% | 11% |

Table 6.3 Tracking errors comparison for LPB and NCC.

## 6.3 Examples of tracking errors

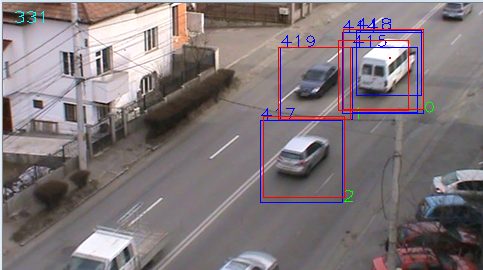


Figure 6.3 Multiple detections for the same vehicle

Figure 6.3 is an example of a vehicle having multiple detections. The cause for this error is the strictness in matching a detection with an existing track. Both the NCC and the LBP can experience this, but we mention that the later exposes more tolerance to these cases. As a vehicle moves through the scene, changing pose and scale, the detector might provide detections centered differently around the vehicle and worst, the size of the detection bounding box is significantly different. The distance of the new detection bounding box around the vehicle and the model stored in the track for the car can be enough to determine not to merge the detection with the track, and thus start a new track. It is also true that we generally noticed these variations in location and size for medium sized vehicles with which the classifier was not properly trained.

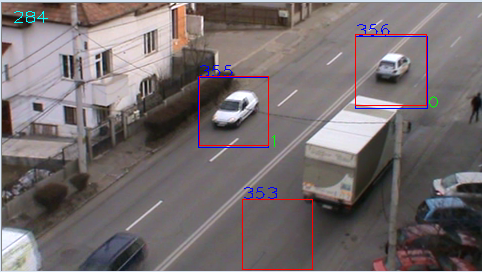


Figure 6.4 Tracker has drifted from the tracked vehicle

In figure 6.4 we have an example of a track that has drifted from the true path of the vehicle. The source of this problem is due to the lack of detections for a initialized track. With no detections arriving from the detector, it is the job of the Median Flow Tracker and the Kalman filter estimator to predict the position of the vehicle. It can happen that the Median Flow Filter has not enough points to track and fail and thus the prediction is left to the Kalman Filter which is not updated and presumes constant velocity in movement, which is not always the case. In these situations the drifted track will get negatively scored each frame until it is removed.

# 7. User manual

## ****7.1 Installation Requirements and Guide****

Since the amount of processing that has to be performed even for a single frame in the video is high, our project has dependencies on native programming technologies (Cuda) and platform specific frameworks (on Windows7 - Asynchronous Agents Library, Parallel Patterns Library). Next there is a list of a set of minimum hardware requirements that we have identified, and dependent software packages that our project is built upon.

Hardware Requirements

* Nvidia Cuda 3.5 capable GPU (we have used a NVidia GT555)
* Minimum Intel Core 2 Duo Hyper threaded (we have used a hyper threaded Core i7)
* At least 6 GB of memory (we had 8 GB of installed RAM memory)

Software Requirements

* Windows 7 on 64 bits
* Nvidia Cuda 3.5 library
* OpenCV 2.3 library
* SVMLight
* Ruby 1.8.3 or higher

Steps to build the project

* Install Cuda 3.5 library from the Nvidia official site – follow the default installation
* Download OpenCV 2.3 library from the Willow Garage official site
  + It is enough to download the executable installer for Windows7, you can also download the source code which you are required to build with CMake but this is not necessary
  + The OpenCV installer will only prompt you to select the extraction path of the the dll’s that it consists of
* For this step you will need to have Visual Studio 2010.
  + Create a property sheet for the project used for VtLib and VtImpl
  + Add the path to OpenCV/build/include to Additional Include Directories
  + Include this property sheet in the rest of the projects
  + For the rest of the projects you should configure the this property sheet by linking the .lib’s to the Linker/Input/Additional Dependencies
* Compile the source code from Visual Studio 2010 in the following order
  + VtLib Project
  + VtImpl Project
  + DatasetExtraction Project
  + TrainingHog Project
* For executing the projects you can either start them from Visual Studio or start the executables from the build or debug folder of the project that Visual Studio has created.

## 7.2 User Guide

In order to use the project you must have a video sequence of a surveillanced road segment conforming to the constraints that we have specified in the paper (video taken from a highly mounted viewpoint and though which the movement if the cars is relatively linear). Having the video which you want to work with you should place it in a specific folder within which the executables will search when specifying video file names. This folder is ./CarParkMonitor/Content/Videos. In this folder you should the video with which we worked with called evo1.avi. We should also mention that in order for the video to be properly read, it should be converted with the ffv1 video codec. We recommend a simple tool called Any Video Converter to convert your video if they are not of the specified format.

After you have copied your video into this folder you must start the Training Set Aquisition process. You do this by starting the DatasetExtraction project from Visual Studio witht the name of the video as a command line or executing the following command from the release or debug folder:

*DatasetExtraction.exe <you\_video\_file\_name>*

After issuing the command a window should appear displaying the first frame of the video. In this step you are required to select a region of interest within which the vehicles should be saved to a data base when they pass by dragging the mouse. The region of interest would also bee chosen having in mind some constraints, namely the vehicles that pass through the selection suffer minimal occlusion and also they should expose the general orientation of the road. Figure 7.1 provides an example of our scene and a proper region of interest as a selection.

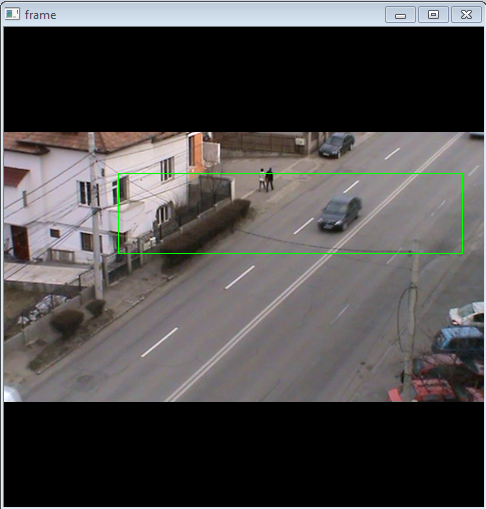
 

Figure 7.1 Example of prompted video window, example of selection

After the Training Set Aquisition process has ended or has been interrupted, the ./CarParkMonitor/Content/Images folder should contain all the images that have been extracted. The next step requires that you have the Ruby Virtual Machine installed, and you need to run the file, which is located in ./Content folder, from the command line or just double click it for execution:

*rename-images.rb*

After completion of the previous step you need to execute the following command from the release or debug folder of the project in command line to extract the hog features from the training set and store them:

*TrainingHog.exe*

The previous step should take approximately a minute. Next you need to train the svm with the extracted features from the training set and collect the results. You should run the following command and ruby file front the ./Content folder.

*hog-train.bat*

*extract-hplane.rb*

After the two command have terminated execution you are able to start the video and observe the performance of the vehicle tracking method. Figure 7.2 is an example of the tracking output which we have obtained.

*VehicleTrackingMonitor.exe <your\_video\_file>*

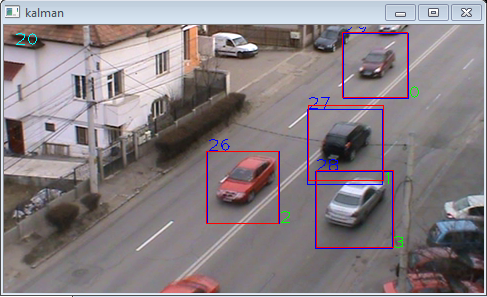


Figure 7.2 Example of tracking

# 8. Conclusions

## 8.1 Summary

The subjects that this paper covers have individually been the topic of many papers because the problems they present are impairing the computer vision domain for years. Throughout this paper we have combined a stack computer vision methods, i.e. image processing and pattern matching techniques, and also machine learning algorithms for achieving vehicle classification, and detection followed by a new tracking method.

We have designed and formalized a pluggable and extensible architecture aimed for object tracking based applications which can be a starting point when we want to evaluate the performance and efficiency of alternative module implementations. We have customized the architecture and implemented the contracts of the modules in the context of evaluating the traffic flow on a monitored road segment by estimating the number of vehicles that pass during the time frame of observation. The primary input for our system is a video sequence taken from a single acquisition device and outputs the vehicle objects in the process of detection and tracking in an annotated window while also keeping count of how many vehicle objects it has registered and scored as belonging to vehicle class.

In general, all machine learning algorithms take as input thousands of data vectors made up of many features, where the number of features might also reach thousands. The first problem that we have encountered and addressed is how to collect and label the images that make up this dataset. In the context of this problem we are proposing to use existing infrastructure without needing to retrieve external sources of data which most of the times are not suitable for your use, but only for the scope they were originally collected for. We have proposed a method to extract the necessary training samples making use of existing infrastructure and maximize precision by making use of scene locality, and ending up with thousands of training images.

Another problem which we had to solve is choosing the features by which to represent a model of a vehicle. It is unavoidable that cars will appear at different scales: their image may consist of just a few pixels, or may be covering the whole screen. Even worse, vehicles will often be occluded, lighting conditions might abruptly change that directly influence the properties of casted shadows. We have imposed a few restrictions on the viewing point and scene complexity in order to deal with these variations in the data and alleviate some problems that were impairing our training set collection, classification, detections and tracking, namely:

* High mounted video acquisition device observing vehicles from a toppish view, this helps us in training set extraction and tracking because:
  + We are making sure that the vehicles exhibit minimal occlusion
  + Vehicles also exhibit small variations in scale as they pass through the frames
* Restrictions on the observed scene helps us construct a robust classifier and tracker
  + Passing vehicles must conform to a fairly linear pattern of motion
  + The variation of possible orientations should vary slightly around a median orientation

After labeling the data, we had to decide which features to extract from the objects. In general, we must find features that express some invariance in the objects, and we decided to extract the HOG features which we expected to be fairly similar to vehicles passing through the scene and expose some robustness to changing light conditions.

Detecting was the next problem that we had to solve. We approached this problem with a classical sliding windows manner of applying the classifier over a pyramid of scales. Of course this method will output multiple detections around the true position of the vehicle which need to be merged.

Tracking is also plagued with the following misfortunes: loss of information caused by projection of the 3D world on a 2D image, noise in images, complex vehicle motion, partial and full occlusions, objects lacking in local discriminance, i.e. cars have matt surfaces and reflective surfaces. In the context of tracking we have described a new tracking algorithm which combines a Median Flow Estimator, which is heavily based on the Lucas-Kanade Optical Flow, and the Kalman Filter, to minimize mutual errors. We have proposed two methods of evaluating these errors, one based on the Normalized Cross Correlation, and the other based on Local Bit Patterns.

Performance is a constraint under which we had to make every decisions, considering that the amount of processing per frame is immense. We have chosen native software technologies in developing our solution (Asynchronous Agents Library and Parallel Patterns Library), and native hardware addressability capabilities (Cuda Library for Nvidia GPUs). Reducing the frame size to an acceptable small dimensions is also a method to improve performance.

The results which we have achieve are fairly good and can compare to other systems developed in other papers, [5] for example where the error in tracking is approximately similar.

## 8.2 Further Development

The main disadvantages of our tracking method are due to the strictness with which the matching between the detection and track is computed, i.e. the locality of features have a height weight in this process. Thus the tracking method is strongly dependent on detection variation around the true location of the vehicle. This means that if we perceive a detection of a vehicle that has initiated a track that does not center the vehicle similarly to the last detection we might initiate a new track under the assumption that we have detected a new vehicle. It is also true that these situations occur on medium to large sized vehicles on which the classifier is not properly trained. An improvement would be to use SIFT or SURF descriptors which discard the locality, but we also expect these methods to have difficulties in achieving desired performance. A further improvement is to perform SIFT and SUFR feature extractions and matching in parallel for tracks.

Another disadvantage of the current system is the impossibility of deploying it in complex scenes, where vehicles expose much more variation in orientation and scale, for example a turning corner. In these situations we might perform training set acquisition and tracking in separate regions that expose fairly small pose and scale variations.

# Bibliography

[1] Robert Laganiere, “OpenCV 2 Computer Vision Application Programming Cookbook”, Packt Publishing, 2011

[2] Cortes Corinna, Vapnik Vladimir N.; “Support-Vector Networks”, Machine Learning Conference, 1995

[3] Sergios Theodoridis, Konstantinos Koutroumbas, “Pattern Recognition”, Academic Press, 2003

[4] Brian Leung, “Component-based Car Detection in Street Scene Images”, Massachusetts Institute of Technology, May 2004

[5] Goo Jun J. K. Aggarwal, “Tracking and Segmentation of Highway Vehicles in Cluttered and Crowded Scenes”, 2009

[6] Birgi Tamersoy, J.K. Aggarwal, “Robust Vehicle Detection for Tracking in Highway Surveillance Videos Using Unsupervised Learning”, 2009

[7] Zdenek Kalal, Krystian Mikolajczyk, Jiri Matas, “Forward-Backward Error: Automatic Detection of Tracking Failures”, International Conference on Pattern Recognition, August 2010

[8] Zdenek Kalal, Krystian Mikolajczyk, Jiri Matas, “Online learning of robust object detectors during unstable tracking”, On-line Learning for Computer Vision Workshop, 2009

[9] Zdenek Kalal, Krystian Mikolajczyk, Jiri Matas, “P-N Learning: Bootstrapping Binary Classifiers by Structural Constraints”, IEEE Conference on Computer Vision and Pattern Recognition, June 13-28, 2010

[10] Jaesik Choi, “Realtime On-Road Vehicle Detection with Optical Flows and Haar-like feature detector”, Computer Science Department University of Illinois at Urbana-Champaign Urbana, 2006

[11] Navneet Dalal, Bill Triggs, “Histograms of oriented gradients for human detection”, IEEE Conference on Computer Vision and Pattern Recognition, 2005

[12] Massimo Piccardi, “Background subtraction techniques: a review”, IEEE International Conference on Systems, Man and Cybernetics, 2004

[13] C. Wren, A. Azarhayejani, T. Darrell, A.P.Pentland, , “Pfinder: real-time tracking of the human body”, IEEE Transactions on Pattern Anaysis and Machine Intelligence, 1997

[14] Gary Bradski, Adrian Kaehler, “Learning OpenCV”, O’Reilly Media, 2008

[15] R. E. Kalman, “A New Approach to Linear Filtering and Prediction Problems”, Journal of Basic Engineering, 1960

[16] Lucas D. Bruce, Kanade Takeo, “An Iterative Image Registration Technique with an Application to Stereo Vision”, Proc. of 7th International Joint Conference on Artificial Intelligence (IJCAI), 1981

[17] Ojala T. Pietikäinen, Harwood, D., “A Comparative Study of Texture Measures with Classification Based on Feature Distributions”, Pattern Recognition, 1996

[18] Ahonen T. Hadid A., Pietikäinen M., “Face Description with Local Binary Patterns: Application to Face Recognition”, IEEE Trans. Pattern Analysis and Machine Intelligence, 2006

[19] Birgi Tamersoy, J.K. Aggarwal, “Counting Vehicles in Highway Surveillance Videos”, International Conference on Pattern Recognition, 2010

[20] Matti Pietikäinen,”Local Bit Patterns” article, http://www.scholarpedia.org/article/Local\_Binary\_Patterns, 2010

[21] Microsoft Asynchronous Agent Library, source: http://msdn.microsoft.com/en-us/library/dd492627

[22] Thorsten Joachims, SVMLight utility, source: http://svmlight.joachims.org/

[23] Precision and Recal, source: <http://en.wikipedia.org/wiki/Precision_and_recall>

[24] Morphological Operators, UTCN Image Processing Laboratory, source: <http://users.utcluj.ro/~raluca/ip_2011/ipl_07e.pdf>

[25] Bilgin Esme, “Kalman Filter” blog post, source: http://bilgin.esme.org/BitsBytes/KalmanFilterforDummies.aspx, 2012

# Appendix A - VtInterfaces

***BgSubtractorBase.h***

#pragma once

#include <opencv2\opencv.hpp>

using namespace cv;

class BgSubtractorBase

{

protected:

char\* windowName;

virtual void init(char\* windowName)

{

instanceNr++;

this->windowName = windowName;

}

public:

static int instanceNr;

BgSubtractorBase()

{

init("subtractor");

}

BgSubtractorBase(char\* windowName)

{

init(windowName);

}

~BgSubtractorBase(){};

virtual void learn(const cv::Mat& frame) = 0;

virtual cv::Mat segment(const cv::Mat& frame) = 0;

virtual Mat getBackground(){return Mat();}

};

***BlobDetector.cpp***

#include "BlobDetector.h"

#include "opencv2/opencv.hpp"

#include <memory>

#include "RectTool.h"

using namespace cv;

static const int MARGIN = 3;

int BlobDetector::instanceNr = 1;

void BlobDetector::init(int minWidth, int minHeight, char\* windowName )

{

instanceNr++;

this->minWidth = minWidth;

this->minHeight = minHeight;

this->minArea = minWidth\*minHeight;

this->windowName = windowName;

}

BlobDetector::BlobDetector(int minWidth, int minHeight)

{

char\* windowName = new char[50];

sprintf(windowName, "detector %d", instanceNr);

init(minWidth, minHeight, windowName);

}

vector<blob> BlobDetector::detect( DetectorParams params )

{

Mat foreground = params.foreground;

Mat frame = params.frame;

Mat maskCopy = foreground.clone();

vector<vector<Point> > contours;

vector<Vec4i> hierarchy;

findContours( maskCopy, contours, hierarchy, CV\_RETR\_EXTERNAL , CV\_CHAIN\_APPROX\_SIMPLE, Point(0, 0) );

vector<blob> foundBlobs;

int size = contours.size();

for( int i = 0; i < size; i++ )

{

Rect rect = boundingRect( Mat(contours[i]) );

Point tl = rect.tl();

Point br = rect.br();

if(rect.area() > this->minArea &&

rect.width > minWidth && rect.height > minHeight &&

tl.x > MARGIN && tl.y > MARGIN &&

(br.x < (frame.cols - MARGIN)) && (br.y < (frame.rows - MARGIN)))

{

blob b;

b.rect = rect;

b.detectorId = i;

b.contour = contours[i];

b.id = ID\_UNDEFINED;

foundBlobs.push\_back(b);

}

}

maskCopy.release();

return foundBlobs;

}

***VideoProcessor.cpp***

#include "VideoProcessor.h"

VideoProcessor::~VideoProcessor(){}

void VideoProcessor::run(FrameProcessor\* frameProcessor, float xspeed )

{

this->capture.open(this->videoPath);

if(!capture.isOpened()){

capture.release();

return;}

double fps = capture.get(CV\_CAP\_PROP\_FPS);

fps = fps/xspeed;

while(capture.read(this->in)){

frameProcessor->process(this->in);

cv::waitKey(fps);

}

capture.release();

}

# Appendix B - VtImplementation

***AvgSubtractor.cpp***

#include "AvgSubtractor.h"

#include "PngSaver.h"

void AvgSubtractor::init( char\* windowName )

{

this->threshold = 20;

this->learningRate = 0.4;

this->closeHolesCount = 1;

int erosionType = MORPH\_CROSS;

int erosionSize = 3;

this->structuringElement = getStructuringElement(

erosionType,

Size( 2\*erosionSize + 1, 2\*erosionSize+1 ),

Point( erosionSize, erosionSize)

);

}

AvgSubtractor::AvgSubtractor():BgSubtractorBase()

{

init("avg-sub");

}

AvgSubtractor::AvgSubtractor( char\* windowName ):BgSubtractorBase(windowName)

{

init(windowName);

}

void AvgSubtractor::learn(const Mat& frame)

{

segment(frame);

}

Mat AvgSubtractor::segment(const Mat& frame )

{

// convert to gray-level image

cv::cvtColor(frame, grayFrame, CV\_BGR2GRAY);

// initialize background to 1st frame

if (background.empty())

grayFrame.convertTo(background, CV\_32F);

// convert background to 8U

background.convertTo(backgroundGray,CV\_8U);

// compute difference between current image and background

cv::absdiff(backgroundGray,grayFrame,foreground);

// apply threshold to foreground image

cv::threshold(foreground, foreground,threshold,255,cv::THRESH\_BINARY);

// accumulate background

cv::accumulateWeighted(grayFrame, background, learningRate, foreground); //can also pass mask to ignore stationary cars

return foreground;

}

cv::Mat AvgSubtractor::getBackground()

{

return this->background;

}

***MogSubtractor.cpp***

#include "MogSubtractor.h"

MogSubtractor::MogSubtractor()

{

init("mog-subtractor");

}

MogSubtractor::MogSubtractor( char\* windowName ):BgSubtractorBase(windowName)

{

init(windowName);

}

void MogSubtractor::learn( const Mat& frame )

{

bgModel(frame, foreground, -1);

}

cv::Mat MogSubtractor::segment( const Mat& frame )

{

bgModel(frame, foreground, -1);

Mat background;

bgModel.getBackgroundImage(background);

cv::erode(foreground, foreground, smallCrossKernel, Point(-1,-1),2);

cv::dilate(foreground, foreground, crossKernel, Point(-1,-1),3);

return foreground;

}

void MogSubtractor::init( char\* windowName )

{

BgSubtractorBase::init(windowName);

bgModel = BackgroundSubtractorMOG2();

bgModel.nShadowDetection = 0;

meanWindowName = new char[50];

sprintf(meanWindowName, "mog-mean-%d", instanceNr);

foregroundWindowName = new char[50];

sprintf(foregroundWindowName,"mog-foreground-%d", instanceNr);

int erosionSize = 2;

elipseKernel = getStructuringElement(

MORPH\_ELLIPSE,

Size( 2\*erosionSize + 1, 2\*erosionSize+1 ),

Point( erosionSize, erosionSize)

);

erosionSize = 2;

rectKernel = getStructuringElement(

MORPH\_RECT,

Size( 2\*erosionSize + 1, 2\*erosionSize+1 ),

Point( erosionSize, erosionSize)

);

erosionSize = 2;

crossKernel = getStructuringElement(

MORPH\_CROSS,

Size( 2\*erosionSize + 1, 2\*erosionSize+1 ),

Point( erosionSize, erosionSize)

);

erosionSize = 1;

smallCrossKernel = getStructuringElement(

MORPH\_CROSS,

Size( 2\*erosionSize + 1, 2\*erosionSize+1 ),

Point( erosionSize, erosionSize)

);

}

cv::Mat MogSubtractor::getBackground()

{

Mat background;

bgModel.getBackgroundImage(background);

return background;

}

***HogClassifier.cpp***

#include "HogClassifier.h"

#include "RectTool.h"

#include "PngSaver.h"

static const int width = 64;

static const int height = 64;

cv::Mat HogDetector::readHPlane(char\* path)

{

auto f = FileStorage(path, FileStorage::READ);

Mat hplane;

f["hplane"] >> hplane;

return hplane;

}

void HogDetector::load()

{

hogGpuDetector = cv::gpu::HOGDescriptor(

Size(width, height),//images size

Size(16,16),//block size

Size(8,8),//block stride

Size(8,8),//cell size

9

);

Mat hplane = readHPlane();

hogGpuDetector.setSVMDetector(hplane);

}

static const int KEEP = 0;

static const int DISCARD = 1;

vector<detection> HogDetector::detect( ClassifierParams& params )

{

Mat frame = params.frame;

vector<Rect> rectDetections;

vector<detection> detectionsToReturn;

int detectionId = 0;

Mat grayClone;

cv::cvtColor(frame, grayClone, CV\_BGR2GRAY);

gpuMat.upload(grayClone);

hogGpuDetector.detectMultiScale(gpuMat,rectDetections);

gpuMat.release();

grayClone.release();

int size = rectDetections.size();

int\* status = new int[size];

memset(status, KEEP, size\*sizeof(int));

for(int i = 0; i < size - 1; i++)

{

for(int j = i+1; j < size; j++)

{

if(status[i] == KEEP && status[j] == KEEP)

{

int min;

if(rectDetections[i].area() < rectDetections[j].area())

min = i;

else

min = j;

Rect intersection = rectDetections[i] & rectDetections[j];

if(intersection.area() > 0.4\*rectDetections[min].area())

{

status[min] = DISCARD;

}

}

}

}

for(int i = 0; i < size; i++)

if(status[i] == KEEP)

{

detection det = {detectionId++, rectDetections[i]};

detectionsToReturn.push\_back(det);

}

delete[] status;

return detectionsToReturn;

}

# Appendix C - DataSetExtraction

***HogFramer.cpp***

#include "HogFramer.h"

#include "Content.h"

#include "Draw.h"

#include "RectTool.h"

#include <math.h>

#include <array>

#include <ppl.h>

#include <concurrent\_vector.h>

#include "AvgSubtractor.h"

#include "MogSubtractor.h"

#include "StableAvgSubtractor.h"

#include "StableMog.h"

#include "PngSaver.h"

using namespace Concurrency;

DrawingControl HogFramer::drawingControl;

vector<Rect> HogFramer::monitoredRegions;

static const double pi = 3.1416;

const char\* frameWindowName = "frame";

static const int width = 64;//80;

static const int height = 64;

static const int cropHeight = 80;

static const int NEG\_INDEX = 0;

static const int POSW\_INDEX = 1;

static const int POSB\_INDEX = 2;

int\*\* HogFramer::pathCounters;

bool isNonCircular(const blob& b)

{

if(b.contour.size() < 20)

return true;

return RectTool::isMostlyCircular(b.contour);

}

HogFramer::HogFramer():FrameProcessor("Hog Extractor")

{

this->blobDetector = new BlobDetector(30, 30, "detector");

HogFramer::drawingControl.drawing = false;

PngSaver::setBasePath("./../CarParkMonitor/Content/Images");

frameCount = 0;

trainingFrames = 100;

posCount = 0;

negCount = 0;

whiteCount = 0;

blackCount = 0;

saveLastRunConfig();

}

void HogFramer::process( const Mat& frame )

{

Mat frameClone = frame.clone();

frameCount++;

if(frameCount == 4)

{

selectMonitorRegions(frame);

initPathCounters(monitoredRegions.size());

for(int i = 0; i < monitoredRegions.size(); i++)

{

RegionExtractor ext;

char\* detectorName = new char[50];

char\* windowName = new char[50];

char\* saveWindowName = new char[50];

sprintf(detectorName, "detector %d", i);

sprintf(windowName, "window %d", i);

sprintf(saveWindowName, "saving %d", i);

ext.id = i;

ext.rect = monitoredRegions[i];

ext.bgSubtractor = new MogSubtractor();

ext.detector = new BlobDetector(30,30,detectorName);

ext.windowName = windowName;

ext.saveWindowName = saveWindowName;

regionExtractors.push\_back(ext);

}

}else

{

for\_each(begin(regionExtractors), end(regionExtractors), [&](const RegionExtractor& ext){

Draw::rect(ext.rect, frameClone);

});

parallel\_for\_each(regionExtractors.begin(), regionExtractors.end(), [&](const RegionExtractor& ext)

{

Mat roi = frame(ext.rect);

if(frameCount < trainingFrames)

{

ext.bgSubtractor->learn(roi);

}else

{

Mat foreground = ext.bgSubtractor->segment(roi);

Mat background = ext.bgSubtractor->getBackground();

DetectorParams detParam;

detParam.foreground = foreground;

detParam.frame = roi;

vector<blob> blobs = ext.detector->detect(detParam);

std::remove\_if(begin(blobs), end(blobs),isNonCircular);

savePositiveAndNegatives(blobs,ext.rect, roi, background, frame, ext.id);

}

});

}

imshow(frameWindowName, frameClone);

frameClone.release();

}

void HogFramer::savePositiveAndNegatives( vector<blob> blobs,const Rect& crop,const Mat& roi, const Mat& roiBackground ,const Mat& frame, int regId)

{

for\_each(begin(blobs), end(blobs), [&](const blob& b)

{

Point cropTopLeft = crop.tl();

Rect blobRect = b.rect;

Point centerOnCrop = RectTool::center(blobRect);

Point centerOnFrame = cropTopLeft + centerOnCrop;

Rect cropRect = Rect(0,0, width, height);

RectTool::toCenter(centerOnFrame, cropRect);

if(RectTool::isContained(cropRect, crop))

{

Point centerOnRoi = RectTool::center(blobRect);

Rect rectOnRoi = Rect(0, 0, width, height);

RectTool::toCenter(centerOnRoi, rectOnRoi);

Mat regionOnRoi = roi(rectOnRoi);

savePositive(regionOnRoi, Mat(), regId);

generateNegativeSamples(cropRect, frame);

}

});

}

void HogFramer::generateNegativeSamples( const Rect& r, const Mat& frame, int regId)

{

static Rect frameRect(0,0, frame.cols, frame.rows);

Rect topLeft = Rect(0,0,width, height);

RectTool::toCenter(r.tl(), topLeft);

Rect topRight(0,0, width, height);

Point p = r.tl() + Point(width, 0);

RectTool::toCenter(p, topRight);

Rect bottomLeft = Rect(0,0, width, height);

p = r.tl() - Point(0, height);

RectTool::toCenter(p, bottomLeft);

Rect bottomRight = Rect(0,0, width, height);

p = r.br();

RectTool::toCenter(p, bottomRight);

unique\_ptr<char> path;

if(RectTool::isContained(topLeft,frameRect))

{

path = nextNegPath(regId);

imwrite(path.get(), frame(topLeft));

}

if(RectTool::isContained(bottomRight, frameRect))

{

path = nextNegPath(regId);

imwrite(path.get(), frame(bottomRight));

}

}

void onMouseClick(int event, int x, int y, int flags, void\* param)

{

switch(event)

{

case CV\_EVENT\_LBUTTONDOWN:

HogFramer::drawingControl.drawing = true;

HogFramer::drawingControl.x = x;

HogFramer::drawingControl.y = y;

break;

case CV\_EVENT\_LBUTTONUP:

if(HogFramer::drawingControl.drawing)

{

int width = abs(HogFramer::drawingControl.x - x);

int h = cropHeight;//abs(HogFramer::drawingControl.y - y);

Rect recti = Rect(HogFramer::drawingControl.x, HogFramer::drawingControl.y, width, h);

HogFramer::monitoredRegions.push\_back(recti);

HogFramer::drawingControl.drawing = false;

}

break;

default:

break;

}

}

void HogFramer::selectMonitorRegions(const Mat& frame)

{

Mat clone = frame.clone();

const char\* name = frameWindowName;

imshow(name, clone);

cv::setMouseCallback(name,onMouseClick);

while(true)

{

for\_each(begin(HogFramer::monitoredRegions), end(HogFramer::monitoredRegions), [&](Rect re){

Draw::rect(re, clone);

imshow(name, clone);

});

if(cv::waitKey(100) > 0)

break;

}

clone.release();

cv::setMouseCallback(name, NULL);

}

# Appendix D - HogFeatureExtraction

***main.cpp***

#include <stdio.h>

#include <iostream>

#include <fstream>

#include <opencv2\opencv.hpp>

#include "Loader.h"

using namespace cv;

using namespace std;

struct RunConfig

{

int width;

int height;

};

RunConfig readLastConfig()

{

RunConfig c;

c.width = 64;

c.height = 64;

return c;

}

int main(int argc, char\*\* argv)

{

RunConfig cfg = readLastConfig();

static const int width = cfg.width;

static const int height = cfg.height;

static const int fvectorSize = 2268;

cv::HOGDescriptor hog(

Size(width, height),//images size

Size(16,16),//block size

Size(8,8),//block stride

Size(8,8),//cell size

9

);

Loader::setPosPathAndCount("./../CarParkMonitor/Content/DatasetExtractions/ext2", 1001);

Loader::setNegPathAndCount("./../CarParkMonitor/Content/DatasetExtractions/ext1",600);

ofstream train;

train.open("./../CarParkMonitor/Content/Assets/to\_test\_positive.txt");

train.clear();

vector<float> descriptor;

auto positives = Loader::getPositiveSamples();

for\_each(begin(positives), end(positives), [&](const Mat& image){

descriptor.clear();

hog.compute(image, descriptor);

train << "+1";

int size = descriptor.size();

for(int i = 0; i < size; i++)

{

int findex = i+1;

train <<" "<< findex <<":"<<descriptor[i];

}

train << endl;

});

train.close();

return 0;

}

# Appendix E - VehicleTracker

***PTracker.cpp***

#include "PTracker.h"

#include "Draw.h"

#include "DrawExtensions.h"

#include <algorithm>

#include "RectTool.h"

#include "BlobDetector.h"

#include "lbp.h"

#include "histogram.h"

#define PROCESS 1

const int XCount = 9;

const int YCount = 9;

const int minTrackedPoints = 10;

const float treshold = 0.3;

static const int bufferSize = 3;

struct trackMatch

{

int detectionId;

float score;

};

struct detectionMatch

{

int trackId;

float score;

};

bool PTracker::shiftBuffers( SubFrame subResult, ClasifierFrame classifierResult )

{

Mat frame = subResult.frame;

Mat foreground = subResult.foreground;

frameRect = Rect(0,0, frame.cols, frame.rows);

Mat grayFrame;

cv::cvtColor(frame, grayFrame, CV\_BGR2GRAY);

auto frameDetections = classifierResult.detections;

frameBuffer.push\_back(frame);

foregroundBuffer.push\_back(foreground);

detectionBuffer.push\_back(frameDetections);

grayFrameBuffer.push\_back(grayFrame);

if(frameBuffer.size() > bufferSize)

{

frameBuffer[0].release();

foregroundBuffer[0].release();

grayFrameBuffer[0].release();

detectionBuffer[0].clear();

frameBuffer.erase(frameBuffer.begin());

grayFrameBuffer.erase(grayFrameBuffer.begin());

foregroundBuffer.erase(foregroundBuffer.begin());

detectionBuffer.erase(detectionBuffer.begin());

}else

return false;

return true;

}

#pragma region kalman predictors

bool PTracker::getKalmanPrediction( track& tr, Rect& predictedRect)

{

if(kalmanFilters.find(tr.id) != kalmanFilters.end())

{

auto filter = kalmanFilters[tr.id];

auto kalmanResult = filter->predict();

auto rect = kalmanResult.asRect();

predictedRect = rect;

tr.model.kalmanRect = rect;

}else

assert(false);

return true;

}

void PTracker::correctKalman( track& tr )

{

if(kalmanFilters.find(tr.id) != kalmanFilters.end())

{

auto filter = kalmanFilters[tr.id];

KalmanInput2D input = {tr.asRecti()};

auto result = filter->correct(input);

auto rect = result.asRect();

tr.model.kalmanRect = rect;

}else

assert(false);

}

void PTracker::forwardKalman( track& tr )

{

if(kalmanFilters.find(tr.id) != kalmanFilters.end())

{

auto filter = kalmanFilters[tr.id];

auto nextPrediction = filter->predict();

KalmanInput2D fakeInput = {nextPrediction.asRect()};

auto fakeResult = filter->correct(fakeInput);

tr.model.kalmanRect = fakeResult.asRect();

}else

assert(false);

}

#pragma endregion

#pragma region unified lucas kanade tracking

void PTracker::beginTracking()

{

allRegisteredPoints.clear();

registeredStatuses.clear();

registeredTracks.clear();

xmedianForTracks.clear();

ymedianForTracks.clear();

scaleForTracks.clear();

trackKey = 0;

}

int PTracker::registerForTracking( track& tr )

{

int id = trackKey++;

auto trackRect = tr.asRecti();

Mat currentForeground = foregroundBuffer[0];

if(RectTool::area(trackRect) < 400 || !RectTool::isContained(trackRect, frameRect))

{

registeredStatuses[id] = false;

tr.lkId = id;

return id;

}

trackRect = trackRect & Rect(0,0, currentForeground.cols, currentForeground.rows);

Mat trackForeground = currentForeground(trackRect);

int xoffset = trackRect.width / XCount;

int yoffset = trackRect.height / YCount;

Point tl = trackRect.tl();

vector<Point2f> trackedPoints;

for(int i = 0; i < XCount; i++)

{

for(int j = 0; j < YCount; j++)

{

int line = yoffset\*j;

int col = xoffset\*i;

uchar value = trackForeground.at<uchar>(line,col);

if( value == 255)

trackedPoints.push\_back(Point2f(tl.x, tl.y) + Point2f(col,line));

}

}

if(trackedPoints.size() < minTrackedPoints)

{

registeredStatuses[id] = false;

tr.lkId = id;

return id;

}

int begin = allRegisteredPoints.size();

allRegisteredPoints.insert(allRegisteredPoints.end(), trackedPoints.begin(), trackedPoints.end());

int end = allRegisteredPoints.size() - 1;

trackEntry entry = {begin, end, id};

registeredTracks[id] = entry;

tr.lkId = id;

return id;

}

void PTracker::performTracking()

{

if(allRegisteredPoints.size() == 0)

return;

vector<Point2f> ftracked;

vector<Point2f> btracked;

vector<uchar> fstatus, bstatus;

vector<float> ferrors, berrors;

cv::calcOpticalFlowPyrLK(

grayFrameBuffer[0],

grayFrameBuffer[1],

allRegisteredPoints,

ftracked,

fstatus,

ferrors

);

cv::calcOpticalFlowPyrLK(

grayFrameBuffer[1],

grayFrameBuffer[0],

ftracked,

btracked,

bstatus,

berrors

);

for(auto it = begin(registeredTracks); it != end(registeredTracks); ++it)

{

int begin = (\*it).second.begin;

int end = (\*it).second.end;

int id = (\*it).second.id;

int sz = end - begin + 1;

float\* xdiffs = new float[sz];

float\* ydiffs = new float[sz];

int successCount = 0;

vector<Point2f> inliers0, inliers1;

for(int i = begin; i <= end; i++)

{

if(fstatus[i] == 1 && bstatus[i] == 1)

{

auto vec = btracked[i] - allRegisteredPoints[i];

float dist = sqrt(vec.x \* vec.x + vec.y \* vec.y);

if(dist < treshold)

{

inliers0.push\_back(allRegisteredPoints[i]);

inliers1.push\_back(ftracked[i]);

Point2f diff = ftracked[i] - allRegisteredPoints[i];

xdiffs[successCount] = diff.x;

ydiffs[successCount] = diff.y;

successCount++;

}else

{

fstatus[i] = 0;

bstatus[i] = 0;

}

}

}

if(successCount > minTrackedPoints)

{

float xmedian = RectTool::median(xdiffs, successCount);

float ymedian = RectTool::median(ydiffs, successCount);

float \*scales = new float[sz\*(sz-1)/2];

int comparisons = 0;

for (int i = begin; i < end; i++) {

for (int j = i + 1; j <= end; j++) {

if (fstatus[i] == 1 && fstatus[j] == 1) {

float dxPrev = allRegisteredPoints[j].x - allRegisteredPoints[i].x;

float dyPrev = allRegisteredPoints[j].y - allRegisteredPoints[i].y;

float dxNext = ftracked[j].x - ftracked[i].x;

float dyNext = ftracked[j].y - ftracked[i].y;

float sPrev = sqrt(dxPrev \* dxPrev + dyPrev \* dyPrev);

float sNext = sqrt(dxNext \* dxNext + dyNext \* dyNext);

if (sPrev != 0 && sNext != 0) {

scales[comparisons] = sNext / sPrev;

comparisons++;

}

}

}

}

float scale = RectTool::median(scales, comparisons);

xmedianForTracks[id] = xmedian;

ymedianForTracks[id] = ymedian;

scaleForTracks[id] = scale;

registeredStatuses[id] = true;

}else

{

registeredStatuses[id] = false;

}

delete[] xdiffs;

delete[] ydiffs;

}

}

bool PTracker::getMedianFlowPrediction(track& tr, Rect& predictedRect )

{

if(registeredStatuses.find(tr.lkId) == registeredStatuses.end())

assert(false, "this should not happen");

if(registeredStatuses[tr.lkId] == false)

return false;

predictedRect = tr.asRecti();

RectTool::move(predictedRect, xmedianForTracks[tr.lkId], ymedianForTracks[tr.lkId]);

float scale = scaleForTracks[tr.lkId];

RectTool::scale(predictedRect, scale, scale);

return true;

}

#pragma endregion

Rect PTracker::mergePredictions(bool lucasSuccess, bool kalmanSuccess, track& tr, Rect& rectLucas, Rect& rectKalman, vector<Mat> grayBuffer , float& bestMatchDist)

{

auto grayFrame = grayBuffer[1];

auto lucasRect = rectLucas & frameRect;

auto kalmanRect = rectKalman & frameRect;

Mat lkSrc = grayFrame(lucasRect);

Mat kalmanSrc = grayFrame(kalmanRect);

if(lucasSuccess)

{

float dist0 = secMatcher->distance(tr, lkSrc);

float dist1 = secMatcher->distance(tr, kalmanSrc);

if(dist0 <= dist1)

{

bestMatchDist = dist0;

return rectLucas;

}

else{

bestMatchDist = dist1;

return rectKalman;

}

}

else

{

bestMatchDist = secMatcher->distance(tr, kalmanSrc);

return rectKalman;

}

}

void PTracker::run()

{

IdGenerator idGenerator(0);

frameCount = 0;

while(true)

{

SubFrame subResult = receive(subtractorBuffer);

ClasifierFrame detectionResult = receive(classifierBuffer);

bool buffersFull = shiftBuffers(subResult, detectionResult);

if(!buffersFull)

continue;

frameCount++;

#if PROCESS

#pragma region setting data for time t

auto detections = detectionBuffer[1];

auto currentFrame = frameBuffer[1];

auto currentGrayFrame = grayFrameBuffer[1];

auto prevFrame = frameBuffer[0];

auto currentForeground = foregroundBuffer[1];

#pragma endregion

#pragma region init tracks from first detections

if(tracks.size() == 0 && detections.size() > 0)

{

tracks.clear();

tracks.reserve(detections.size());

for\_each(begin(detections), end(detections), [&](detection& d){

createTrack(d, idGenerator, currentGrayFrame);

});

continue;

}

#pragma endregion

#pragma region predict track positions for time t

auto lkoutput = currentFrame.clone();

deleteExitedTracks();

beginTracking();

for(auto it = begin(tracks); it != end(tracks); it++)

registerForTracking(\*it);

performTracking();

for(auto it = begin(tracks); it != end(tracks); it++)

{

Rect kanadePrediction, kalmanPrediction;

bool lucasSuccess = getMedianFlowPrediction(\*it, kanadePrediction);

bool kalmanSuccess = getKalmanPrediction(\*it, kalmanPrediction);

float minDist = 999999;

auto finalPrediction = mergePredictions(lucasSuccess, kalmanSuccess, \*it, kanadePrediction, kalmanPrediction, grayFrameBuffer, minDist);

it->assign(finalPrediction);

it->predictionDist = minDist;

}

#pragma endregion

std::map<int, trackMatch> trMatches;

std::map<int, detectionMatch> detMatches;

matcher->begin();

secMatcher->begin();

for\_each(begin(tracks), end(tracks), [&](track& tr){

auto dit = detections.begin();

auto dend = detections.end();

for(;dit != dend; ++dit)

{

float score = matcher->match(tr, \*dit, currentGrayFrame);

if(score < 0)

continue;

float dist = secMatcher->match(tr, \*dit, currentGrayFrame);

if(dist > secMatcher->goodMaxDist)

{

continue;

}

trackMatch trMatch = {dit->id, score};

detectionMatch dMatch = {tr.id, score};

if(detMatches.find(dit->id) == detMatches.end())

{

trMatches[tr.id] = trMatch;

detMatches[dit->id] = dMatch;

}else if(detMatches[dit->id].score < score)

{

auto exmatch = detMatches[dit->id];

trMatches.erase(exmatch.trackId);

trMatches[tr.id] = trMatch;

detMatches[dit->id] = dMatch;

}

}

});

#pragma region update track models and kalman

for\_each(begin(tracks), end(tracks), [&](track& tr){

if(trMatches.find(tr.id) != trMatches.end())//matched detection with track

{

auto m = trMatches[tr.id];

matcher->inferModel(tr, detections[m.detectionId], currentGrayFrame);

secMatcher->inferModel(tr, detections[m.detectionId], currentGrayFrame);

validators[tr.id].tick(true);

correctKalman(tr);

}else

{//detection not found for track

float max = secMatcher->maxSimilarityDist;

float dist = tr.predictionDist;

if(dist < max)

validators[tr.id].tick(true);

else

validators[tr.id].tick(false);

forwardKalman(tr);

}

});

#pragma endregion

#pragma region init new tracks from unmatched detections

for\_each(begin(detections), end(detections), [&](detection& mockdet){

auto it = detMatches.find(mockdet.id);

if(it == detMatches.end()){

auto inited = createTrack(mockdet, idGenerator, currentGrayFrame);

trackMatch m = {mockdet.id, 1};

trMatches[inited.id] = m;

}

});

#pragma endregion

#pragma region drawing\_results

auto fclone = currentFrame.clone();

DrawExtensions::drawDetections(detections, fclone);

DrawExtensions::drawTracks(tracks, fclone, Scalar(255,0,0));

for\_each(begin(tracks), end(tracks), [&](track& tr){

Draw::rect(tr.model.kalmanRect, fclone, Scalar(0,0,255));

});

std::stringstream str;

str << carCount;

Draw::text(str.str(), Point(10,20), fclone, Scalar(255,255,0));

imshow("kalman", fclone);

fclone.release();

lkoutput.release();

#pragma endregion

#endif

send(syncBuffer,1);}}