## 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.

## 5.1 Language, Technology, Frameworks

*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, as stated in [21].

One should take a look at the improvements and new standards adopted for this version, and which the Visual Studio 2010 compiler include 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

*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

*Asynchronous Agents Library (AAL) and Parallel Pattern Library (PPL)*. According to [24] 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

*SVMLight*. This is a small executable downloaded from [23] offering implementation of Support Vector Machines (SVMs) in C.

*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.

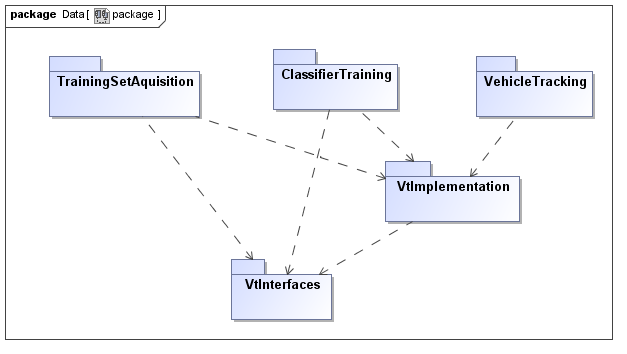


Figure 5… System package diagram

The packages and the main concerns are depicted in the following table.

|  |  |
| --- | --- |
| Packages | Concerns and Responsibilities |
| * 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 |

5.2.1 VtInterfaces

Figure 5… presents in detail the inner packages and classes that the VtInterfaces (Vehicle Tracking Interfaces) package organizes.

## 

Figure 5… 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 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.

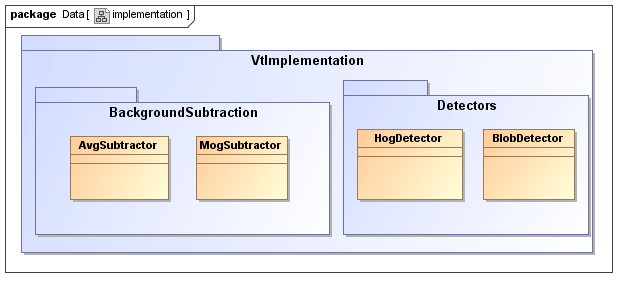


Figure 5… Inner packages and classes of VtImplementation package

As shown in figure 5… 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…. 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.

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… show 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… Code for computing the Running Average Background Model

Figure 5… Code for computing the MOG Background Model

Figure 5… 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… show 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… Code for GPU HOG detection

5.3.3 TrainingSetAqusition

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 save images centered and containing these groups we must apply some minimal tests be sure we are not saving other objects or move 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, 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… is the class diagram for this subsystem.

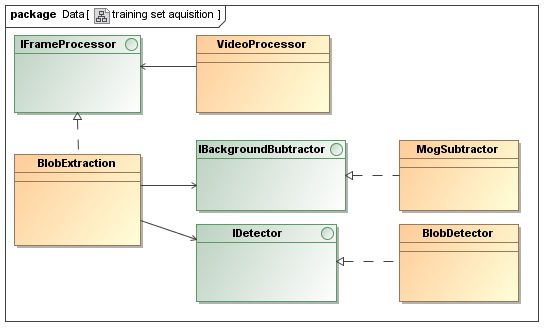


Figure 5… Training Set Acquisition class diagram

5.3.4 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, which are also vectors, together. 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.

5.3.5 Tracking

5.3.5.1 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… 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 [24] 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 queue. 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… shows the classes that inherit the **agent** class from AAL and figure 5… depicts the data flow between the instances of the classes and the asynchronous communication buffers. All classes that inherit the agent class has to override the virtual method *run*() , where we have to implement the logic for each component. The **PVideo** (Parallel Video) class is responsible for reading the frames from the video on a different thread and sending clones of frames along the unbound buffers that lead to the PSubtractor and PDetector instances.

The **PSubtractor** instance is delegating the background subtraction operation to the encapsulated IBackgroundSubtractor implementer, which in this case is performing the Mixture of Gaussian method.

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. The operations background subtraction and detection are performed in parallel.

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 established 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 foreground buffer and detection buffer. Once these buffers both have a foreground mask and detection for the next frame it consumes these entries from the input unbound buffers and forwards the data structures to the tracking algorithm.

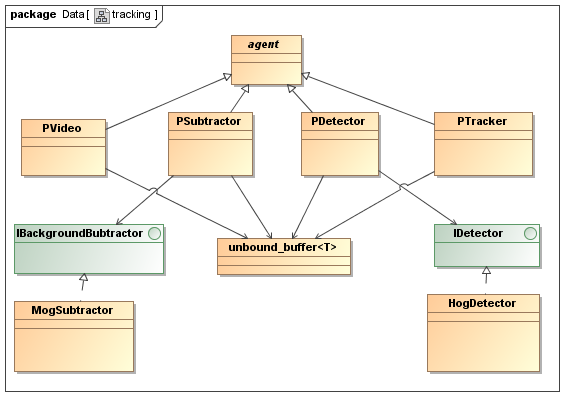


Figure 5…, Class diagram of tracking subsystem



Figure 5… Data flow diagram of tracking subsystem

5.3.5.1 Tracking algorithm details

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 is the best estimation 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. The next figure illustrates the interface of our class adapted for the case of tracking the position of objects.

Figure 5… Kalman Filter declaration

The **Median Flow Tracker** is implemented in the PTracker class and makes heavy use of the Lucak-Kanade Point Tracker which OpenCv also exposes:

Figure 5… OpenCv call to track points from one grayscale image to 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…



Figure 5… 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. The upper figure, figure 5…, show the functions that you need to call to perform tracks estimation 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… . 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… Simplified code for tracking