Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Radioelectronics

Object Detection in Video Signal

Ladislav Kršek

Supervisor: prof. Ing. Pavel Zahradník, CSc.
Field of study: Communications and signal processing
August 2020

Acknowledgements

Děkuji panu profesorovi Pavlu Zahradníkovi za vedení mé diplomové práce.

Declaration

I declare that I completed the presented thesis independently and that all used sources are quoted in accordance with methodical instructions that cover the ethical principles for writing academic thesis.

In Prague, 13. August 2020

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských prací.

V Praze, 13. August 2020

Abstract

This diploma thesis deals with object detection in a video signal, implemented for Raspberry-Pi platform using *C* programing language. Object detection is based on "Histogram of Oriented gradients" (HOG) algorithm and "Support Vector Machine" (SVM) classification. The introductory contains description of the Raspbery-Pi platform and describes HOG and SVM algorithms. The following chapter contains code design specification and the information about the implementation to reach the desired detection.

Keywords: Detection, videosignal, HOG, SVM, Raspberry-Pi, camera, C, Matlab

Supervisor: prof. Ing. Pavel Zahradník, CSc.
katedra telekomunikační techniky,
Technická 1902/2,
Praha 6

Abstrakt

Tato diplomová práce se zabýva detekcí objektu ve videosignálu, implementovaného pro platformu Raspberry-Pi v programovacím jazyce *C*. Detekce objektu funguje na základě algoritmu "histogramu orientovaných gradientů" (HOG) a klasifikátoru "support vector machine" (SVM). V úvodní části obsahuje popis platformy Raspbery-Pi a algoritmů HOG a SVM. V následující časti je zdokumentována implementace a popis dosažení požadované detekce objektu.

Klíčová slova: Detekce, videosignál, HOG, SVM, Raspberry-Pi, kamera, C, Matlab

Překlad názvu: Detekce objektu ve videosignálu

Contents

Figures

Tables

Chapter 1

Introduction

Nowadays the automation is a trend, which is noticeable in almost every branch, even in those that were recently only domain of humans. One of the many parts of automation can also be an object or person detection in a video signal, which is plenteously utilized in manufacturing, automotive, surveillance or entertainment industry. Therefor it is required to create and improve the object detection systems, so they can perform its purpose as best they can.
The goal of the thesis is to implement the detection algorithm histograms of oriented gradients for the Raspberry-Pi platform, which records on-line video signal at 60 frames per second. Since a Raspberry-Pi has convenient proportions and can be powered from an external battery, as the tracked objects were chosen vehicles. Therefore it could be used as monitoring of traffic capacity of certain areas or as the counter of traffic density. However, the object of interest can be easily modified by training the classifier on different data.
The detection pipeline uses algorithm histograms of oriented gradients, which describes each local object shape in the captured frame by the distribution of gradients of edge directions. The values produced by the algorithm are called features and are provided to classification algorithm which compares them with the features in the database. The shape with the most similar features is proclaimed as the local object present in the frame.
The next chapter 2 describes the Raspberry-Pi platform, used model for the implementation, the camera hardware used for capturing the video signal and the application programming interface needed for handling incoming frames. The chapter 3 contains information about the *Histogram of Oriented Gradients* and *Support Vector Machine* algorithms, which are needed for successful detection. The chapter 4 provides information about implementation details of the processing pipeline. Testing of the application is described in chapter

5 and the thesis summary is written in the chapter 6.

Chapter 2

Raspberry Pi Platform

Raspberry-Pi is a series of single-board computers developed by *Raspberry Pi Foundation*. It is mainly used for teaching purposes and research projects. Its advantages are low cost, portability and its own operation system *Raspberry Pi OS*, which is free operating system based on *Linux Debian* distribution, optimized for the *Raspberry Pi* hardware. The hardware uses Micro SD card as the on-board storage and offers connection for external peripherals via USB ports. *Raspberry-Pi* also allows connection of external monitor via HDMI cable and the camera hardware via *CSI-2 (Camera Serial Interface 2)*. The hardware contains system on chip *Broadcom BCM2711*, which has integrated *Quad-core Cortex-A72 (ARM v8) 64-bit, 1.5 GHz* CPU and the *VideoCore IV* GPU. Specification of all the hardware present on the used *Raspberry Pi 4* model can be found at [3].

2.1 Used aliases in next sections

The following aliases are used in next chapters:

- *Camera hardware*
  - The actual *Raspberry Pi Camera v2.1* hardware that is connected to the *Raspberry Pi* board.
  - See the next section 2.2 for more information.
- *Camera component*
  - MMAL API component, which is *C* language structure of the type *MMAL_COMPONENT_T*. The structure represents an object

    that is created and destroyed during run-time of the process and is used to control the *Camera hardware*, receives the frames from the driver and forwards them for further processing within the running process.
  - See the section 2.3 for more information.

2.2 Camera Hardware

The used camera hardware is *Raspberry Pi Camera v2.1*, which contains **Sony IMX219** sensor. The sensor captures images using method rolling shutter. As mentioned earlier the camera hardware is connected via *CSI-2 (Camera Serial Interface 2)*. The *Raspberry Pi* offers MMAL API for convenient controlling of the camera hardware [2].

2.2.1 Rolling Shutter

The rolling shutter is a method of image capture or video recording. Each frame is not captured at the same time instant by scanning the entire scene, but scanning the scene rapidly, either vertically or horizontally. The full frame is then constructed from all the rows or columns. The disadvantage of this method is that it can create distortions, when capturing fast moving objects or rapid flashes of light. On the other hand the image sensor can gather protons for longer period of time, thus increasing the sensitivity. Rolling shutter is mostly used in digital cameras using CMOS sensors [12].

Exposure Time

Exposure time is duration during which the image sensor is exposed to light. The sensor elements detect photons and built up charge, that is converted via Analog-to-digital-converter (ADC) to the digital numbers. The digital numbers are saved for each pixel. The camera hardware performs two basic operations:

- Read the row/column
  - The row/column is put into the whole frame (matrix)
- Reset the row/column

- The row/column is reset to initial values so that the camera hardware can react on changes of the scene

The exposure time is controlled by altering the periods of both operations. The periods don't have to be the same, but they are synchronized. [6]
The **minimum exposure time** is: $exp_{min} = N \cdot read_{time}$, where N is number of row/columns and $read_{time}$ is the limitation of the hardware to read one row/columns.
The **maximum framerate** is then: $fps_{max} = \frac{1}{exp_{min}}$

Data Processing

The CPU is running Linux based OS and the GPU uses Real-time-OS ThreadX. The communication between both processors is provided by kernel driver VideoCore Host Interface Queue (VCHIQ) [7]. The VCHIQ is then used via Multi-Media Abstraction Layer (MMAL) API, which provides camera hardware control for user applications. (see section 2.3 for more information about MMAL API)
The memory of the Raspberry Pi is split between both processors [6]. See figure 2.1.

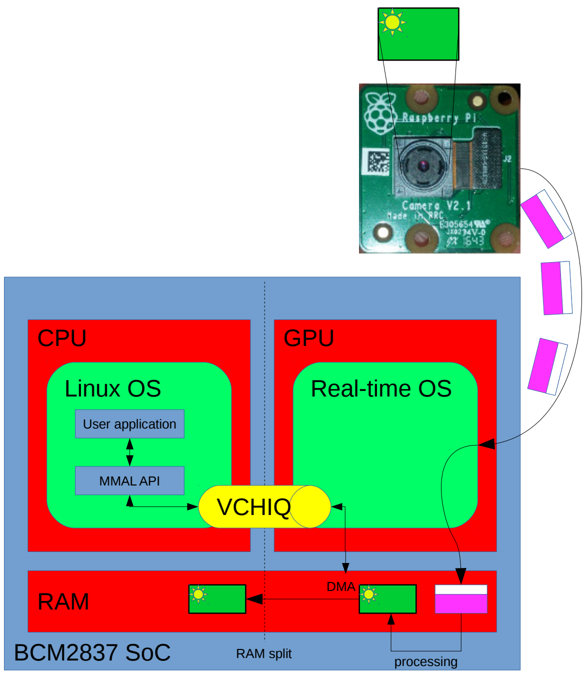


Figure 2.1: Data Processing on the BCM2837

The data processed in the following way [6]:

- The configured camera sensor streams lines of the frame (rows/columns) over the CSI-2 interface to the GPU
- The GPU constructs the frames from the streamed lines and performs post processing tasks:
  - transposition, resizing, digital-gain etc.
- The *user application* requests the recorded frames using the MMAL API
- MMAL API request the recorded frames using the *VCHIQ* driver
- The GPU performs direct memory access (DMA) transfer from its RAM partition to the CPU's RAM partition
- The GPU sends information to CPU via *VCHIQ* that the transformation is completed
- The MMAL API is informed and passes the information to the *user application*

2.3 Multi-Media Abstraction Layer (MMAL) Application Programming Interface (API)

The Raspberry Pi Camera hardware is controlled using the MMAL API [8] developed by Broadcom Europe Ltd. It is written in *C* programing language and offers solutions for following tasks:

- Controlling the camera hardware
- Encoding/decoding video and audio
- Encoding/decoding images

The API is based on the concept of components, ports and buffer headers. The components provide ports (output/input), which produce/receive buffer headers containing the actual data and auxiliary data needed for processing. The components are objects (structures) that are created during rune-time of the program and provide such services as mentioned earlier. See section 4 for code implementation detail.

2.3.1 MMAL API Components

Components are *C* structures of type *MMAL_COMPONENT_T*, that can either produce or process the data contained in buffer headers. The processed data can be passed to another component or simply saved on the filesystem etc. The example of a component could be *camera component*, which controls and receives frames from the camera hardware (see figure 2.1) and returns them at the output port as buffer headers. Another example could be *video encoder component*, which receives incoming plain frames at the input port and returns encoded frames at the output port. The components are created using MMAL API function *mmal_component_create*(), which accepts the component name as an argument and pointer to allocated *MMAL_COMPONENT_T* structure. The predefined components have defined names in API header *mmal_default_components.h*. [8]

2.3.2 MMAL API Ports

Ports are structures *MMAL_PORT_T*, that are created by the components automatically. They contain pointer to *MMAL_ES_PORT_T*, which is structure that defines format of the port. Example of the port format could be resolution of the captured video or the encoding bit rate. Format of output ports are usually set automatically, when the component has sufficient information about the data it produces. The format of input ports has to be set by user via function *mmal_port_format_commit()*. Two MMAL API components can be connected together via connecting output port of the first one and input port of the second one. The procedure is dona via function *mmal_port_connect()*. [8]

Setting Port Parameters.
The parameters are defined by integer indexes. Since the *C* language does not allow overloading, the actual parameter values are represented as binary data and later casted to the specific type based on the ID of the parameter. The structure *MMAL_PARAMETER_HEADER_T* encapsulates general parameters. Setting of the ports parameters is done using function *mmal_port_parameter_set()*. Eventually the parameters can be set using functions that are specific for the type. For example *mmal_port_parameter_set_int32()*.

Enabling the Ports.
Ports are enabled via function *mmal_port_enable()*, with pointer to callback function as the parameter. The callback function is invoked, when the component is done processing the buffer header. When the ports are connected together to callback parameter to *mmal_port_enable()* function has to be NULL. Connected output ports also have assigned callback, which just simply passes the buffer header to the associated input port. This way the port just simply calls provided callback and does not have differentiate between those cases. [8]

2.3.3 Buffer Headers

Buffer headers are structures *MMAL_BUFFER_HEADER_T* used to exchange data between components. The structure contains address of the actual data being transferred. The reason for this approach is ability to provide any type of user data. The buffer headers are created from the pools (structures *MMAL_POOL_T*). The pools allocate fixed number of buffer headers and contain a queue (structures *MMAL_QUEUE_T*), which is basic FIFO queue that provides thread-safe implementation. The queues are simply maintained by the functions *mmal_queue_get()* and *mmal_queue_put()*. [8]

Chapter 3

Description of algorithms used in detection

3.1 Histograms of Oriented Gradients (HOG)

3.1.1 Feature Descriptors

Feature descriptors (features) are information, used in image processing, which contain relevant data about the image or part of the image (patch). Features may be specific structures in the image such as points, edges or objects. They are produced by algorithms, which can be highly dependent on the application.

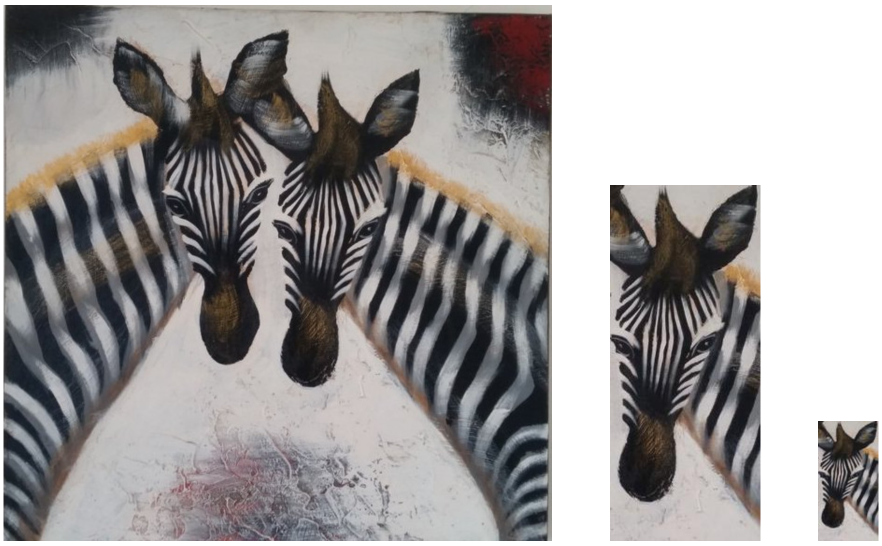
3.1.2 Introduction to HOG

The histogram of oriented gradients (HOG) is a feature descriptor, which counts occurrences of gradient orientation for each image patch. The basic idea is that the local object shape in the image can be described by the distribution of gradients of edge directions. The whole image is divided into cells and a histogram of gradient directions is created from all the pixel within the cell. The final feature descriptor is the vector of all the histograms. Since the HOG operates only in local cells, it is less prone to photometric and geometric transformations, except to object orientation. [1]

3.1.3 Algorithm Overview

Preprocessing

HOG feature descriptor used for pedestrian detection described in the original article [1] is calculated on a 64×128 (width$\times$height) patch of an image. The first step is to select the image patch and resize it to 64×128.
According to [1] the gamma and color normalization have only a small effect on performance, so they will not be considered. The image patch can be transformed to gray scale, which reduces the amount of gradient computation, but according to [1] reduces slightly the performance of the detector. The illustration of the preprocessing part is shown in figure 3.1.



(a) : Original image of size 574×587

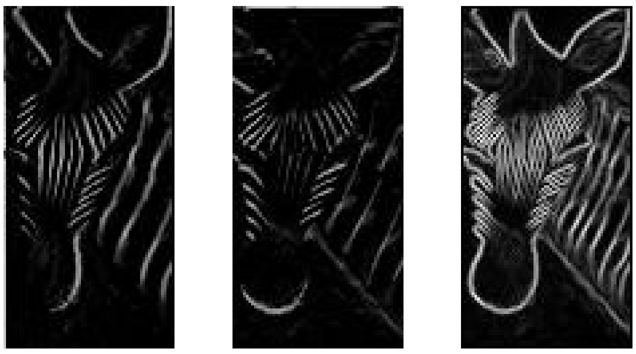
(b) : Image patch of size 381×161

(c) : Image patch resized to 128×64

Figure 3.1: Images used to demonstrate HOG

Gradient Computation

The Cartesian gradients are computed by filtering the image patch with following kernels: $\begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$ and $\begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$, which produce two matrices with x-direction gradients and the y-direction gradients. Afterwards the gradients are converted to polar coordinates, producing two matrices with magnitudes and directions of the gradients. [1]
If the image patch has multiple channels, the gradients are computed for each channel and the one with the largest magnitude is selected as the pixel's gradient vector. The visualizations of the gradients are shown in the figure 3.2.



(a) : Gradient x-direction

(b) : Gradient y-direction

(c) : Gradient magnitudes

Figure 3.2: Gradients of the image

Computation of Histograms in 8×8 Cells

The image patch is divided into cells and a histogram of gradients is calculated for each of them. This approach makes the representation more robust to noise. In [1] the HOG was used for human detection, therefor they selected size of the cell as 8×8. This size is big enough to capture interesting features, such as face and the top of the head.

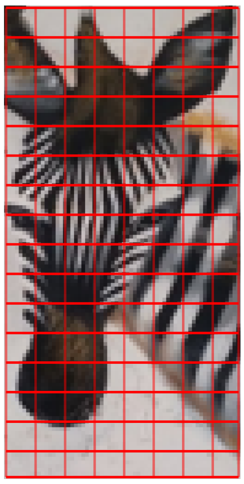


Figure 3.3: Visualization of the 8×8 cells

The histogram is created from gradient orientations placed in the bins. The bins are evenly spaced over $[0, \pi]$ or $[0, 2\pi]$. If the $[0, \pi]$ interval is used, the signs of the gradients are ignored. In the original paper [1] is stated that usage of the whole $[0, 2\pi]$ interval with doubled number of bins (to keep the same bin interval) decreases performance. The number of bins used is 9, giving each one of them $\frac{\Pi}{9} rad$ for the $[0, \pi]$ interval.
Each gradient orientation in the selected cell is bilinearly interpolated between two adjacent bins [9]. The weights are simply the gradient magnitudes. Computation of all histograms in the image patch of size 64×128 with cell proportions 8×8 pixels and 9 histogram bins, produces $9 \cdot 8 \cdot 16 = 1152$ numbers. The visualization of the cells is shown in figure 3.3.

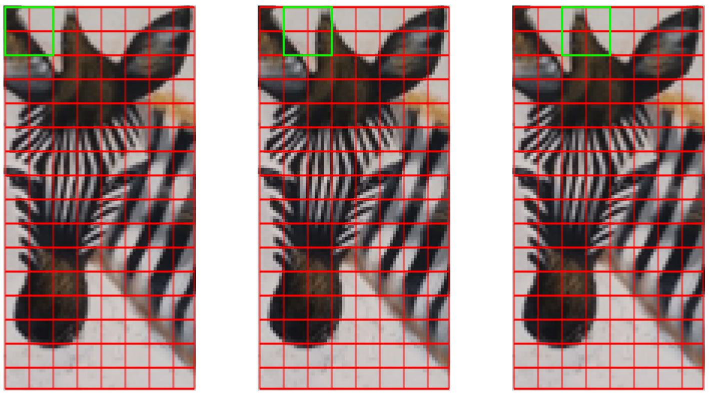
Normalization of gradients in greater blocks

The gradients are sensitive to illumination and foreground to background contrast. Darker images have lower magnitudes of the gradients. If some part of the image is darker then the other one, the histograms in each cell have different intensity, even though they should have the same weights. The way to reduce the contrast is to normalize the histograms in different blocks. The easiest block as described in [1] is rectangular. The cells are simply grouped into greater blocks of size 16×16 pixels, which equals to size 2×2 cells created in section 3.1.3. The blocks overlap by 50% as shown in figure 3.4. Each block is characterized by histograms of the cells in it. For aforementioned size each of them is represented by vector of length 36. The normalization can be done in various ways. According to [1] the one of the most effective is

L2-norm:

$$\mathbf{v}_{norm} = \frac{\mathbf{v}}{\sqrt{||\mathbf{v}||_2^2 + \epsilon^2}} \tag{3.1}$$

where $\mathbf{v}$ is vector containing all the histograms in the block and ϵ is regularization.



(a) : The first block

(b) : The second block

(c) : The third block

Figure 3.4: Image patch rectangular block normalization

HOG Features for the Complete Image

After normalization in the blocks the total number of features is $9 \cdot 4 \cdot 15 \cdot 7 = 3780$ for the image patch of size 64×128 pixels, 8×8 pixel size of the cell, 16×16 pixel size of the blocks and 9 bins in each histogram. This vector represents features of the patch and is sent for further processing.

3.2 Classification of Data

3.2.1 Introduction

In order to determine, whether the image patch contains desired object, the features are provided to a classifier. The classifier estimates to which class the input feature vector belongs. The number of classes can be variable, but the object detection requires only two classes:

- "object is present" (represented by value 1)
- "object isn't present" (represented by value -1)

In the original paper [1] they used the linear and Support Vector Machine classifier to distinguish them.

3.2.2 Support Vector Machine (SVM)

SVM is learning algorithm, that analyzes the data and classifies them. In the learning mode the algorithm is provided with training data examples, which are identified with one of the classes. Based on the examples the algorithm builds the model. The model is then used in the classification mode, which assigns new data to one of the classes. The input data examples are represented as vectors in N-dimensional space. The goal of the learning mode is to find the hyperplane (figure 3.5, "hyperplane0"), that separates the classes in the way, that the distance between them is maximized. [11]. In the classification mode the new data fall to one side of the hyperplane.

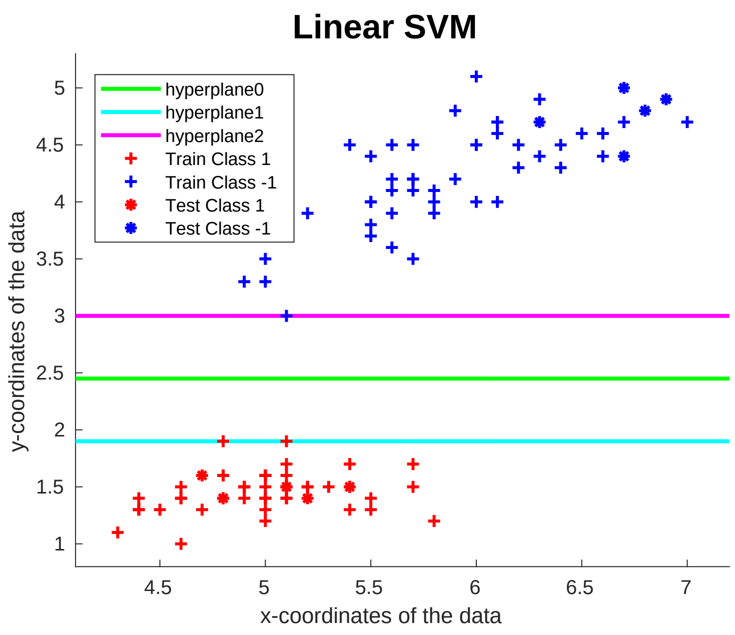


Figure 3.5: Linear SVM 2-D (2 features) example

Support Vectors

The support vectors are the data that are the closest to the hyperplane and influence it's actual position and orientation. These vectors are used to maximize the margin of the classification. When some of those vectors are removed, the hyperplane changes its position. The vectors lie on the "hyperplane1" and "hyperplane2" in figure 3.5.

Linear SVM

The given training data examples are set of vectors : $(\mathbf{x}_i, y_i)$, where the $\mathbf{x}_i$ is the real vector representing each data (features of the image) and y_i is either 1 or -1 depending on class, the data belong to. The goal of the learning mode is to find the hyperplane, which separates both classes with maximum margin. Hyperplane in N-dimensional Euclidean space is defined:

$$\mathbf{w} \cdot \mathbf{x} + b = 0 \tag{3.2}$$

where $\mathbf{w}$ is the normal vector to the hyperplane.
If the data are linearly separable, the two parallel hyperplanes can be found, that separate two classes of data as demonstrated in example figure 3.5. The distance between those hyperplanes is called the *margin*. They are defined as:

$$\mathbf{w} \cdot \mathbf{x} + b = 1 \tag{3.3}$$
$$\mathbf{w} \cdot \mathbf{x} + b = -1 \tag{3.4}$$

The *margin* is equal to $\frac{2}{||\mathbf{w}||}$, where the $||\mathbf{w}||$ is norm of the normal vector to the hyperplanes. All the training data have to lie outside of the margin, which is defined as:

$$\mathbf{w} \cdot \mathbf{x}_i + b \geq 1, \text{ for } y_i = 1 \tag{3.5}$$
$$\mathbf{w} \cdot \mathbf{x}_i + b \leq -1, \text{ for } y_i = -1 \tag{3.6}$$

which can be rewritten as:

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1 \geq 0 \tag{3.7}$$

The maximal margin between the pair of hyperplanes is found by minimizing $||\mathbf{w}||^2$ with constraints 3.7. The mentioned optimalization problems can be solved with Lagrange multipliers. The Lagrangian is given by:

$$L = \frac{1}{2}||\mathbf{w}||^2 - \sum_{i=1}^{n} \alpha_i \big(y_i(\mathbf{w}\cdot\mathbf{x}_i + b) - 1\big) \tag{3.8}$$

$$L = \frac{1}{2}||\mathbf{w}||^2 - \sum_{i=1}^{n} \alpha_i y_i(\mathbf{w}\cdot\mathbf{x}_i + b) + \sum_{i=1}^{n} \alpha_i \tag{3.9}$$

where α_i, $i = 1,..,n$ are Lagrange multipliers and n is equal to number of training examples. The L has to be minimized with respect to $\mathbf{w}$ and b. The 3.9 is a convex quadratic programming problem, because the objective function $\frac{1}{2}||\mathbf{w}||^2$ is convex itself and the constraint points which satisfy 3.7 also form a convex set [11]. This means that the Lagrangian dual problem:

$$\max_{\alpha} \inf_{\mathbf{w},b} \Big(\frac{1}{2}||\mathbf{w}||^2 - \sum_{i=1}^{n} \alpha_i \big(y_i(\mathbf{w}\cdot\mathbf{x}_i + b) - 1\big)\Big) \tag{3.10}$$

$$\alpha_i \geq 0,\ \ i = 1,..,n \tag{3.11}$$

can be solved as Wolfe dual problem, which is following:

$$\max_{\mathbf{w},b,\alpha} \Big(\frac{1}{2}||\mathbf{w}||^2 - \sum_{i=1}^{n} \alpha_i \big(y_i(\mathbf{w}\cdot\mathbf{x}_i + b) - 1\big)\Big) \tag{3.12}$$

$$\nabla_{\mathbf{w},b}(\frac{1}{2}||\mathbf{w}||^2) - \sum_{i=1}^{n} \alpha_i \nabla_{\mathbf{w},b}\Big(\big(y_i(\mathbf{w}\cdot\mathbf{x}_i + b) - 1\big)\Big) = 0 \tag{3.13}$$

$$\alpha_i \geq 0,\ \ i = 1,..,n \tag{3.14}$$

Solving the equation 3.13 :

$$\Big[\mathbf{w},\ 0\Big] + \Big[-\sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i,\ \sum_{i=1}^{n} \alpha_i y_i\Big] = \Big[0,\ 0\Big] \tag{3.15}$$

which gives:

$$\mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i \tag{3.16}$$

$$\sum_{i=1}^{n} \alpha_i y_i = 0 \tag{3.17}$$

Substituting 3.16 and 3.17 into 3.9 gives:

$$L = \frac{1}{2}||\sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i||^2 - \sum_{i=1}^{n} \alpha_i y_i (\sum_{j=1}^{n} \alpha_j y_j \mathbf{x}_j \mathbf{x}_i) - b\sum_{i=1}^{n} \alpha_i y_i + \sum_{i=1}^{n} \alpha_i \tag{3.18}$$

$$L = -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \mathbf{x}_j + \sum_{i=1}^{n} \alpha_i \tag{3.19}$$

The function L has to maximized with constraints:

$$\sum_{i=1}^{n} \alpha_i y_i = 0 \tag{3.20}$$

$$\alpha_i \geq 0,\ \ i = 1,..,n \tag{3.21}$$

Solving the Optimalization Problem

The quadratic programming problem 3.19 can be solved in *Matlab* using function *quadprog* from Optimalization toolbox. The 3.19 has to be rewritten into [10]:

$$\min_{\alpha} \frac{1}{2}\alpha^T \underline{H}\alpha + \mathbf{1}^T\alpha \text{ such that } \begin{cases} -\underline{1}\cdot\alpha & \leq \mathbf{0} \\ \mathbf{y}\cdot\alpha & = 0 \end{cases} \tag{3.22}$$

where $\underline{H} = (\mathbf{y}\cdot\mathbf{y}^T) * (\mathbf{x}\cdot\mathbf{x}^T)$, $*$ is element wise multiplication ,

$\mathbf{1}^T = \begin{bmatrix}1, & 1, & .. & 1\end{bmatrix}$, $\mathbf{0}^T = \begin{bmatrix}0, & 0, & .. & 0\end{bmatrix}$, $\underline{1} = \begin{bmatrix}1, & 0, & .. & 0 \\ 0, & 1, & .. & 0 \\ : & : & .. & : \\ 0, & 0, & .. & 1\end{bmatrix}$

The returned vector α from function *quadprog* is then used to find the weights $\mathbf{w}$ using equation 3.16. In addition the value b is computed from equations 3.3 and 3.4 using support vectors (they lie on one of the hyperplanes, depending on the class, so they have to meet one of the equations). The support vectors have α_i equal to some positive value. Other vectors have proportionally smaller value, because they don't contribute to the actual position of the hyperplanes.

The result of the simple data with two features, 90 training data and 10 testing data is shown in the figure 3.5. The data are linearly separable. Implemented *Matlab* code can be seen in enclosed CD (see section 4.6.2 for the directories hierarchy)

Chapter 4

Code Design Specification

4.1 Assembling MMAL API Components

The processing pipeline consists of four MMAL API components (*MMAL_COMPONENT_T*) connected together. The implementation supports two modes: *Render and Encoder* (see section 4.3 for more information and figures 4.1 and 4.2, which visualize the connection of the components). The *Camera component* is provided by the MMAL API. This component produces buffer headers from incoming frames captured by the *camera hardware* and forwards them on two output ports. The second component is *Encoder component*, which encodes the captured video into H.264 format and saves the captured frames on the file-system. The *HOG component* is user written, which takes incoming frames, computes HOG feature descriptors (see section 3.1 about HOG) and marks the detected object. The last component is video *Render*, which displays on-line the captured video.

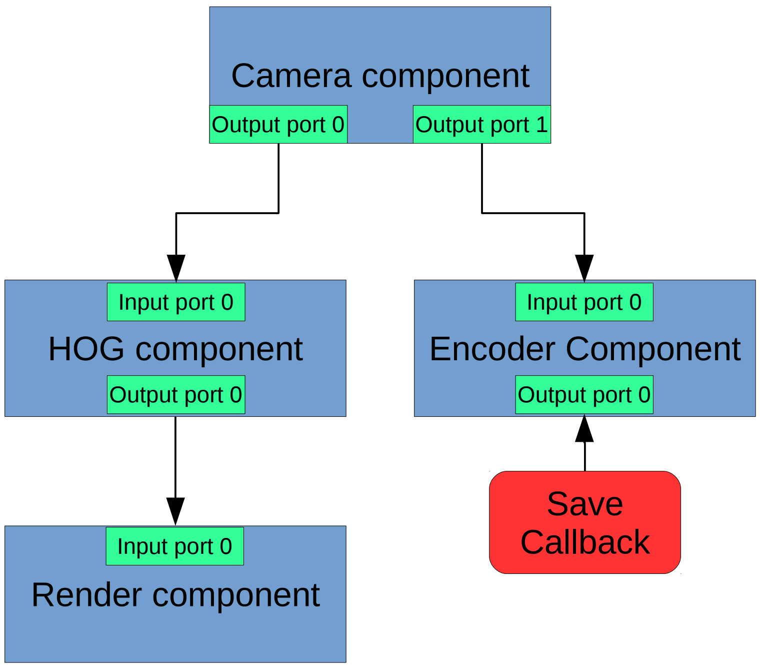


Figure 4.1: Connection of components for the *Render Mode*

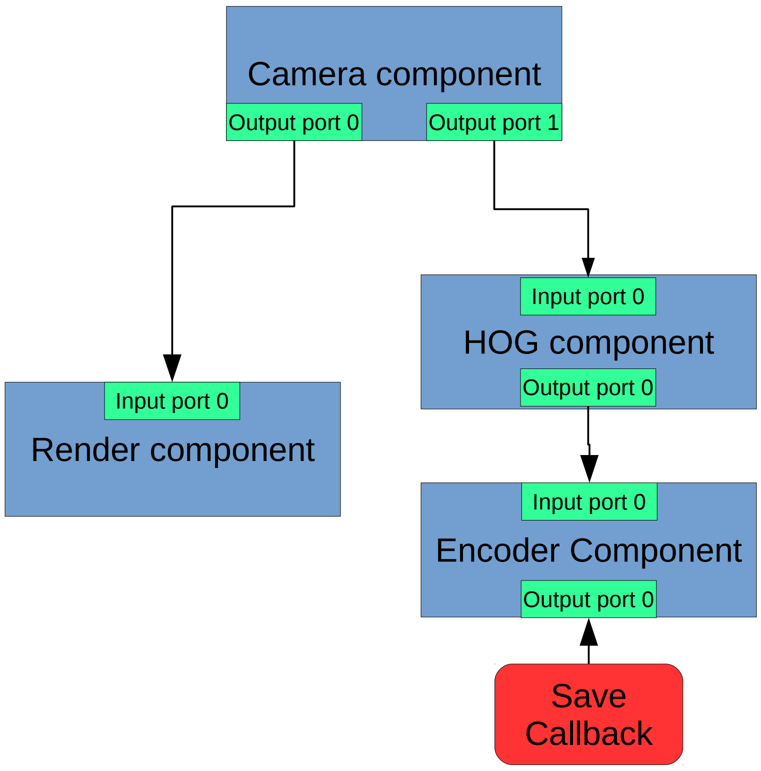


Figure 4.2: Connection of components for the *Encoder Mode*

4.1.1 The Camera component

The purpose of this component is to control and set-up the camera hardware, receive the incoming frames and send them to other connected components for the further processing. The implementation of the component is already provided by the MMAL API itself. The *Camera Component* structure *MMAL_COMPONENT_T* is created using function *mmal_component_create()* with parameter *name* equal to macro *MMAL_COMPONENT_DEFAULT_CAMERA*. The creation of the component is handled in the source file **CameraComponent.c**.

Camera initialization

Initializing camera control port. The camera component contains *control port* of the type *MMAL_PORT_T*, which is used for setting basic parameters of the camera. Minimal settings are:

- Camera ID
  - ID of the camera (zero index if only one camera is connected)
  - Defined by the structure *MMAL_PARAMETER_CAMERA_NUM_T*
- Video or still image resolution
  - Defined by the structure *MMAL_PARAMETER_CAMERA_CONFIG_T*

Parameters above are set using the function *mmal_port_parameter_set()* with the camera control port and the pointer to item **hdr** contained in the structure. The control port is enabled by using the function *mmal_port_enable()* while providing the pointer to control port and the control callback. The callback is triggered, when the parameters of the camera change during runtime (e.g. video resolution, frames per second, etc.).

Initializing output ports. The component uses three output ports, which are connected to input ports of other components based on current mode:

- Camera port
  - *Render Mode*
    - This port produces frames forwarded to *Encoder* component for further processing
  - *Encoder Mode*
    - This port produces frames forwarded to *HOG* component for further processing
- Preview port
  - *Render Mode*

        - Outputs frames forwarded to *HOG* component for further processing
    - *Encoder Mode*
        - Outputs frames forwarded to *Render* component for further processing
- Still port
    - Outputs still images.
    - This port is not used in the implementation.

Camera and preview ports need additional information about the format of the outputs. The format parameters are contained in the structure *MMAL_PORT_T*. The required ones are:

- encoding
    - The desired encoding of the output frames
    - used: *MMAL_ENCODING_I420*
- Video resolution
- Video cropping
- Frames per second

The values that have been set are confirmed by the function *mmal_port_format_commit()*.

Setting additional camera parameters. Additional parameters of the camera in the implementation are:

- Defined by structure *MMAL_PARAMETER_RATIONAL_T*
    - Saturation
        - Defined by the ID *MMAL_PARAMETER_SATURATION*
    - Sharpness
        - Defined by the ID *MMAL_PARAMETER_SHARPNESS*
    - Contrast
        - Defined by the ID *MMAL_PARAMETER_CONTRAST*
    - Brightness
        - Defined by the ID *MMAL_PARAMETER_BRIGHTNESS*
- Defined by int32_t
    - Rotation
        - The parameter is set because the camera hardware is mounted in the rack upside-down.
        - Defined by the ID *MMAL_PARAMETER_ROTATION*

All the mentioned parameters are set using the pointer to camera control port and functions *mmal_port_parameter_set_rational()* or *mmal_port_parameter_set_int32()* respectively.

4.1.2 The Encoder component

The purpose of this component is to encode the capture video into the *H.264* format. The implementation of the component is already provided by the MMAL API itself. The *Encoder Component* structure *MMAL_COMPONENT_T* is created using function *mmal_component_create()* with parameter *name* equal to macro *MMAL_COMPONENT_DEFAULT_VIDEO_ENCODER*. The creation of the component is handled in the source file **EncoderComponent.c**. The Encoder component has one input port, which receives the video frames and one output port. The output port has set function callback, which is invoked when the component is done with encoding of the current buffer header. The purpose of the callback is to save the data into output file, which is opened by using function *fopen()* before assigning to port.

Initializing of input port

The format of the input port is determined after connecting the camera output port. Therefore it doesn't have to be set.

Initializing of output port

The output port needs additional information about the format. The format parameters are contained in the structure *MMAL_PORT_T*. The required ones are:

- encoding
    - Set to enum MMAL_ENCODING_H264
- bitrate
    - Set to values depending on video resolution
- buffer_size and buffer_num
    - Both set to predefined recommended values

The values that have been set are confirmed by the function *mmal_port_format_commit()*.

Initialization of buffer headers pool

Encoder component needs allocated memory for buffer headers (*MMAL_BUFFER_HEADER_T*), which contains the video frames. The memory is allocated using the function *mmal_port_pool_create()*. The pool of buffer headers is then provided into the output port callback function. The pool is provided via item *userdata* contained in the *MMAL_PORT_T structure*.

Enabling the Encoder output port

The output port of the encoder has to be enabled using the function *mmal_port_enable()*. The parameter also takes the pointer to the function callback described below.

Encoder output port callback. The callback has signature:
void (*)(MMAL_PORT_T, MMAL_BUFFER_HEADER_T*)*.
The callback function checks whether the buffer header contains valid data and saves them to provided file using the function *fwrite()*. Before returning the callback has to release the used buffer header back to the pool using the function *mmal_buffer_header_release()*. As mentioned earlier the callback data are provided using the item *userdata* contained in the *MMAL_PORT_T structure*. Implemented callback requires two variables:

- Pointer to the file (data type *FILE*) obtained by function *fopen()*
- Pool of buffer headers

Both of the variables are encapsulated into the structure *EncoderCallback-Data_t* defined in header file **Constants.h**. The structure is then casted into *struct MMAL_PORT_USERDATA_T* and passed to the output port callback.

4.1.3 The Render component

The purpose of this component is to display on-line the captured video and detected objects when the *Render Mode* is enabled. This component has only one input port and doesn't have any output ports (the incoming buffer headers with video frames are just displayed and then automatically released back to the internal buffer headers pool). The *Render Component* structure *MMAL_COMPONENT_T* is created using function *mmal_component_create()* with parameter *name* equal to macro *MMAL_COMPONENT_DEFAULT_VIDEO_RENDERER*. The creation of the component is handled in the source file **RendererComponent.c**.

Render input port initialization

The only parameter set for input port is provided in structure *MMAL_DISPLAYREGION_T*, which contains information about display size and position. Implementation supports the same size of the window as is the video resolution.

4.1.4 The HOG component

The HOG component is custom MMAL API component. For general information about creation of the custom MMAL API component see the section 4.2.1. For the specific implementation of the *HOG* component see the section 4.2.2.

4.2 Implementation of the HOG component

4.2.1 Custom MMAL API component information

The custom component can be created and registered into MMAL API using function *mmal_component_supplier_register()* with some prefix and function which will create the component. The function has to have signature: *MMAL_STATUS_T(*)(const char*, MMAL_COMPONENT_T*)*

Custom component private data

The component structure *MMAL_COMPONENT_T* has private item *MMAL_COMPONENT_MODULE_T* which is used to preserve essential data during lifetime of the component. The custom component can define its own items of this structure.

Creating input and output ports

The component has to allocate its own memory for the input and output ports. The function for this purpose is *mmal_ports_alloc()*, which associates the desired number of ports with the component.

Setting the actions of the ports

Each port requires functions that are triggered when certain actions appear. The functions are contained in the structure *MMAL_PORT_PRIVATE_T*. The items of the structure used in *HOG component* are following:

- *pf_set_format*
  - This function is used when the user calls the function *mmal_port_format _commit()*.
- *pf_disable*
  - This function is used when the user calls the function *mmal_port_disable()*.
- *pf_flush*
  - This function is used when the user calls the function *mmal_port_flush()*.
- *pf_send*
  - This function is used when the user calls the function *mmal_port_send _buffer()*.
- *pf_parameter_set*
  - This function is used when the user calls the function *mmal_port_parameter _set()*.

Registering the action of the component

The component needs to register an action using function *mmal_component_action_register()*, which is triggered when the component calls the function *mmal_component_action_trigger()*. The signature of the function is *void(*)(MMAL_COMPONENT_T*)*. The registered action is run in a separate thread.

Destroying the component

The component structure has also an item which stores the function for destruction. The function is triggered, when *mmal_component_destroy()* is called with the address of the component.

4.2.2 HOG component implementation

The custom MMAL API component is provided in source file **SecondComponent.c** and the implementation of the object detection based on HOG is located in **HOG.c**. The component suits as the interface for the object detection core. During creation, it spawns threads and synchronizes them during processing phase. Each thread has predefined limits of the frame and does the HOG object detection inside these limits.

Component creation

Assigning port actions. The component first allocates data needed by the MMAL API and assigns functions to the port actions, which are the following:

- Port action: *pf_set_format*
  - The function checks that the format encoding is MMAL_ENCODING_I420, which represents UYV420 color format
- *pf_disable*
  - If the function is called for the first time, it instructs the threads to exit, joins them and calls the function, which is saved on the item *pf_flush*
- *pf_flush*
  - The function obtains the input or output buffer headers queue (depending on the port) and sends all the remaining buffer headers to the connected port.
- *pf_send*
  - The function triggers the assigned action (processing of the incoming frame)

Private component data. After assigning the port actions, the private data structure of the component is allocated. The structure contains following data:

- Data needed by the threads (arrays with size equal to number of threads). See the section 4.2.2
    - Data type *thread_t*
    - Data type *thread_data_t*
        - Structure defined in header **HOG.h**. Contains important information needed by each thread. (see 4.6.1 for detailed information about this structure)
    - Data type *sem_t*
        - Semaphore that informs thread that it can continue with the processing pipeline
        - See the sequence diagram 4.3
    - Data type *sem_t*
        - Semaphore which is used by the thread, when it is done with part of the processing pipeline
        - See the sequence diagram 4.3
    - Data type *limits_simple_t*
        - Structure defined in header **HOG.h**. Contains total frame limits for each thread. Initialized during component creation (see 4.6.1 for detailed information about this structure)
    - Data type *limits_all_t*
        - Structure defined in header **HOG.h**. Contains array of patches limits for each thread. Initialized during component creation (see 4.6.1 for detailed information about this structure)
    - Data type *detection_data_t*
        - Structure defined in header **HOG.h**. Contains detection information for each thread. Initialized during component creation (see 4.6.1 for detailed information about this structure)
    - Data type *visualize_data_t*
        - Structure defined in header **HOG.h**. Contains information about position of the object in the frame. (see 4.6.1 for detailed information about this structure)
- Shared memory between threads
    - Pointer to data type *float*
        - Memory containing all the histograms for the frame. Allocated during component creation.
    - Pointer to data type *float*
        - Memory containing all the features for the frame. Allocated during component creation.
    - Pointer to data type *uint8_t*
        - Memory containing boolean value whether the patch from the current frame contains detected object
    - Pointer to data type *uint8_t*
        - Memory containing accumulated value of each patch, which is used for preventing some of the false positives (see the section 4.2.2 for more information how the values are used)
    - Pointer to data type *MODEL*

    - The model created by *light_svm* library. (see section 4.5.1 for more information about the library)
- Data needed by the MMAL API component
  - Data type *MMAL_QUEUE_T*
    - The buffer header queue needed by the input port. Created during component creation.
  - Data type *MMAL_QUEUE_T*
    - The buffer header queue needed by the output port. Created during component creation.

Providing registered actions. The component provides the function for processing the incoming frames (see section 4.2.2 for more information about main thread processing)

Initializing limits of all threads. At first the whole frame resolution is divided equally between all the processing threads so they do not overlap, using the function *dealThreadLimits()*.

Initializing patches positions. Each thread maintains patches which are bound by the limits dealt in previous step. The patches are of the size 64×64 pixel. The positions of all the patches for each thread are determined by the function *init_limits()*, which also take parameter that sets their overlapping. The presence of an object in the patch is determined by the trained *support vector machine* (see section 3.2.2 about SVM)

Initializing detection and visualization data. Initialization is done via functions *init_detection_data()* and *init_visualization_data()*. For detailed information about both structures see section 4.6.1.

Loading the light_svm model. The model is loaded using functions *read_model()* and *add_weight_vector_to_linear_model()*. For more information about both functions, see section 4.5.1 about *svm_light* library.

Initializing and spawning threads. Addresses of previously allocated data are provided to structures *thread_data_t*, which are then passed to function *pthread_create()*, used to spawn threads. To assure that all the threads were successfully initialized, the component waits for all the threads to confirm success by posting to the semaphore.

Component creation failure. If any of the allocation/ initialization fails during the HOG component creation, the destruction functions is called and the functions returns status indication failure.

Component destruction. The function that destroys the component frees all the allocated memory during component creation, instructs the threads to exit and joins them.

Component processing

As mentioned earlier the HOG processing is done when the component triggers the registered function (*main thread*), which happens for each frame. It takes the buffer header from the queue using the function *mmal_queue_get()*. The address of frame pixel values is provided to all threads and they are instructed to start the computation. The function consecutively waits for threads to finish the computation and instructs them continue three times in order to assured thread safety and synchronization (for more details see section 4.2.2 about thread processing). For the sequence diagram of the component processing and thread processing see figure 4.3.

Visualizing detected object. The main thread evaluates the data received from the other threads by checking, which patches contain detecting object. The patches which contain the object are highlighted by the box, which is made by alternating the pixels on the edge of the patch.

Filtering out standalone false positives. The classification of the objects doesn't have 100% accuracy. To avoid some of the misclassified objects each patch accumulates current number of positive samples. The *accumulated value* is increased each time the object is detected and decreased when the object is not present in the patch. The *accumulated value* has lower and upper boundaries to limit the memory. When the object is detected the sum of *accumulated values* of neighboring patches is computed. If the sum is greater than the specified threshold, the patches (from the same neighborhood) which contain the object in the current frame are highlighted. The processed patches are than marked to prevent from further processing. See flow chart digram in figure 4.6.

Sending the processed frame to the output port. After highlighting the detected objects the buffer header is replicated to the output queue using the function *mmal_buffer_header_replicate()*. At the end the used buffer header is released into its pool.

Thread processing pipeline

Processing function of each thread is divided into three phases:

- Computation of histograms
  - Each thread has defined fixed boundaries in the frame. The frame boundaries of one thread doesn't overlap with boundaries of other ones. In this boundaries the threads compute histograms of gradients. Each histogram represents gradients of cells with 8×8 pixel size. All the histograms are stored in the shared memory array. Each thread writes only into its own part of the array. This way thread safety is assured. After computation, each thread notifies the main thread using the semaphore and waits for the notification.
- Computation of features
  - The features are computed from the histograms by taking four neighboring cells and normalizing the vector (see section 3.1.3 about histogram normalization for more information). On the edges of the boundaries the required histograms are shared between neighboring threads. Because of that, the threads are synchronized from the main thread in the previous step, to ensure that all the histograms are computed before computing the features. All the features are stored in the shared memory array and each thread writes into its own part. After computation, each thread notifies the main thread using the semaphore and waits for the notification.
- Detecting the objects
  - Each thread has already computed boundaries for each patch of size 64×64 pixels. The patches overlap to provide better detection density. For each patch the thread reads features from the shared buffer and provides them to *light_svm* library (see section 4.5.1), which classifies whether the patch contains the object. The classification is done using function *classify_example_linear()*. The function returns value of type *double*, which represents distance from the separation hyperplanes (see section 3.2.2 for more information about Support Vector Machine). Based on the distance the thread decides whether the object is present. Because the patches limits reaches out of the thread frame boundaries, they are synchronized in the previous step to ensure that the neighboring thread has already finished computation of the features. After going through all the patches, the thread notifies and waits for the next frame to arrive.

The thread iterates aforementioned steps until the main thread changes value of the shared flag. The implementation is provided in source file **HOG.c** in function *thread_job()*. See flow chart diagram in the figure 4.4.

4.3 Main process

The main process is responsible for creating, connecting and destroying all the components. It is implemented in source file **Main.c**. The compiled binary accepts arguments with following switches:

- "-t"
  - The switch is followed by unsigned integer value defining the video capture duration (in milliseconds)
  - Default value is 5000
- "-m"
  - The switch is followed by unsigned integer value defining whether the detected objects are highlighted in the video render or saved to file using the Encoder component.
  - 0 (default): *Render Mode*: The detected objects are highlighted in Render component
  - else: *Encoder Mode*; The detected objects are highlighted in saved file via Encoder component

See the flow chart diagram in figure 4.5.

4.4 Flow chart and sequence diagrams

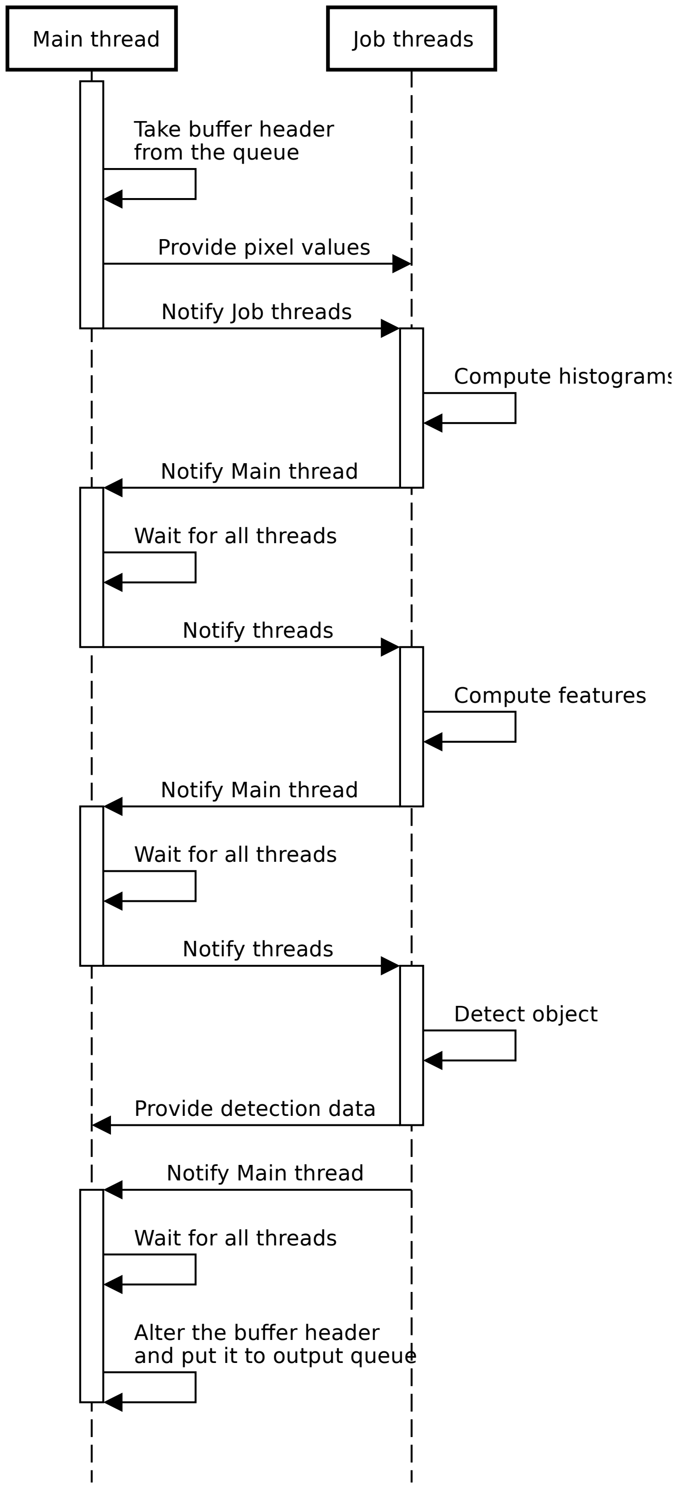


Figure 4.3: Sequence diagram of the thread synchronization

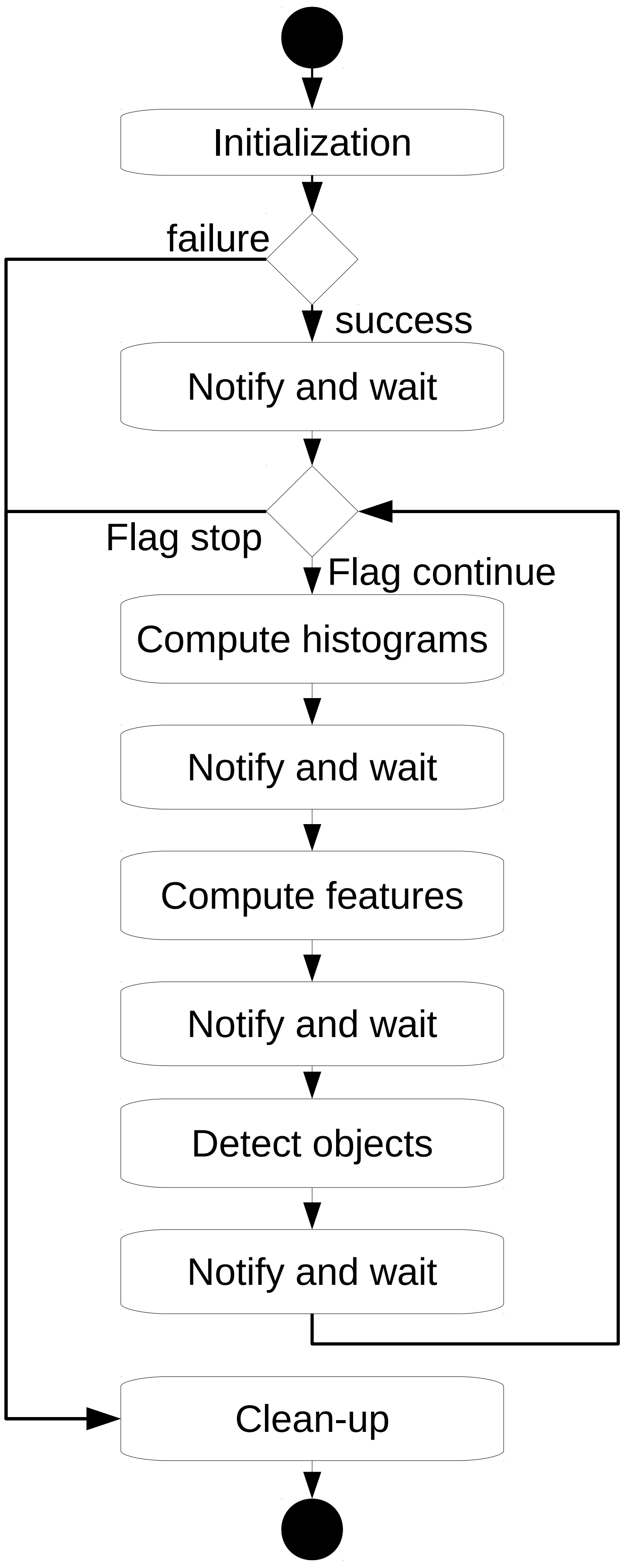


Figure 4.4: Flow chart diagram of the thread processing pipeline

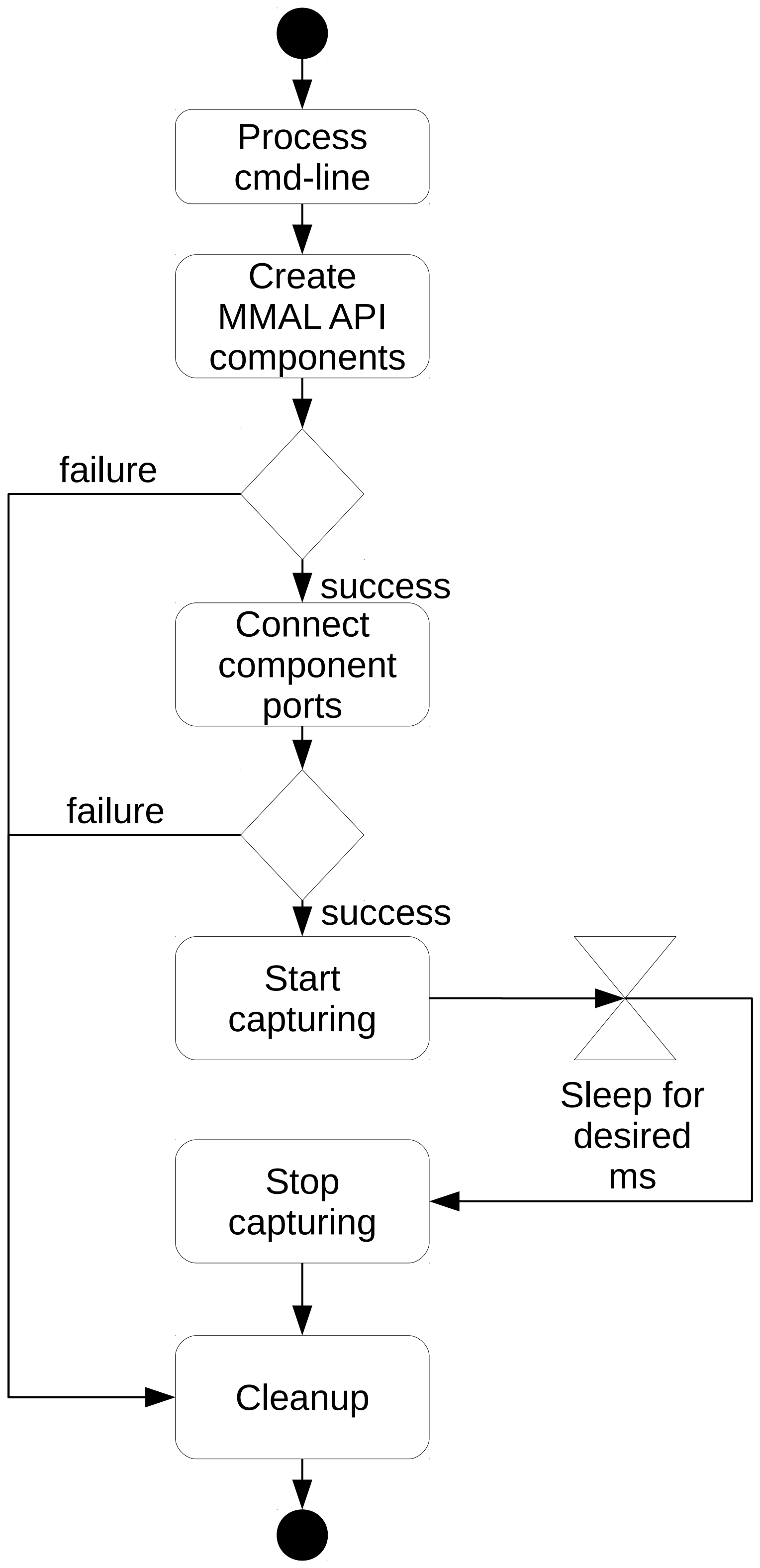


Figure 4.5: Flow chart diagram of the main process

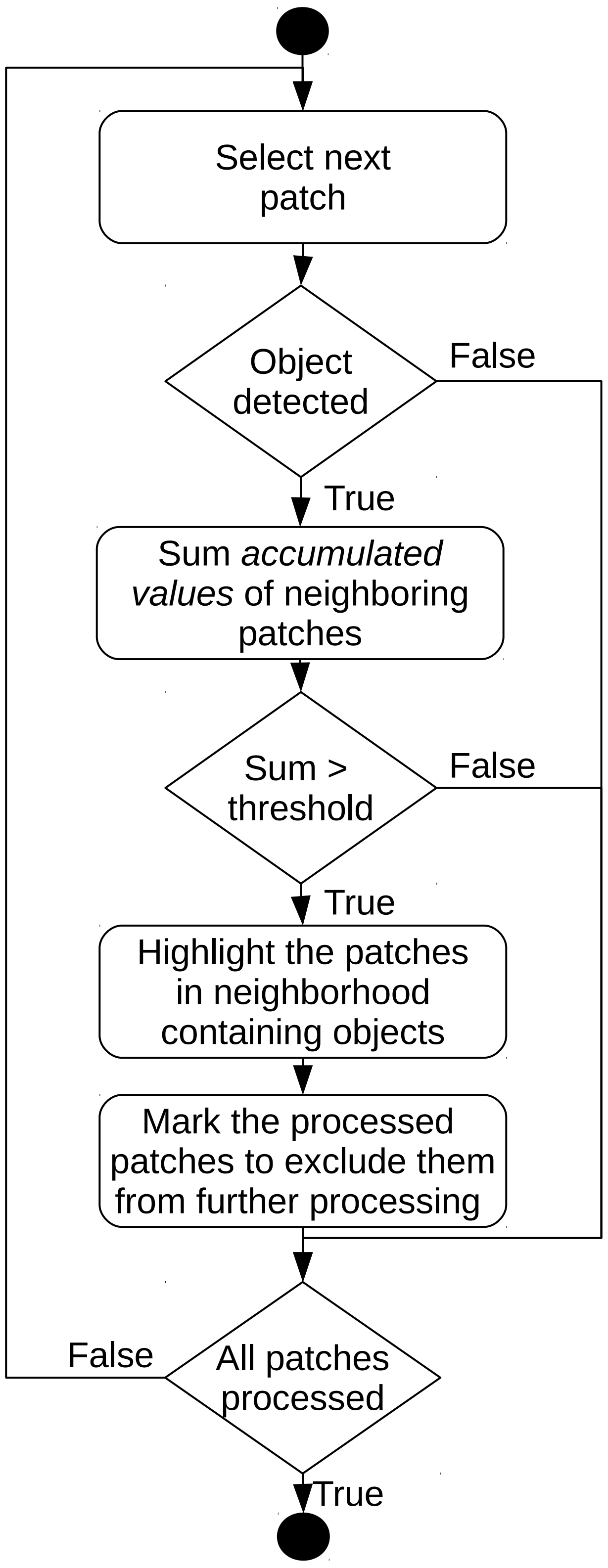

Select next patch
Object detected
False
True
Sum *accumulated values* of neighboring patches
Sum > threshold
False
True
Highlight the patches in neighborhood containing objects
Mark the processed patches to exclude them from further processing
False
All patches processed
True

Figure 4.6: Flow chart diagram of visualizing the detected objects

4.5 Classification

4.5.1 SVMlight Library

Basic information

The effective implementation of the *support vector machine* provides *svm_light* library [5]. The library provides two binaries:

- *svm_learn*
    - Creates classification model based on provided training data. The binary creates text model file, which contains training kernel parameters, dimension size of features and the values of support vectors.
- *svm_classify*
    - Classifies the given test data. The binary loads specified model from the file-system and based on the model classifies given testing data. As the result, outputs number of misclassified samples and the error rate.

Both training and testing data are loaded from the text file in the following format [5]:

<target> <feature>:<value> <feature>:<value> ... <feature>:<value>

where:

- <target> equals -1 or 1 (depending on the class that the example belongs to)
- <feature> is integer index of the feature vector dimension. Indexing starts from value 1.
- <value> is a float value of the feature vector at this index

Entry with <value> 0 can be omitted. SVMlight also provides a dynamically linked library *libsvmlight.so*. The library provides following api for classification:

- Function *read_model()*
    - Loads the trained model from the file-system into structure *MODEL*;
- Function *add_weight_vector_to_linear_model()*
    - Adds linear weights into the loaded *MODEL* structure. Needed only for linear kernel.
- Function *classify_example_linear()*
    - Classifies the given data based on the trained model (provided in structure *MODEL*).

Usage of libsvmlight library in the implementation

The functions *read_model()* and *add_weight_vector_to_linear_model()* are used in the HOG component during the creation. The model is then provided to all the threads. Each thread initializes its own structure *DOC*, which is needed by the function *classify_example_linear()*. The structure contains classification parameters and the feature vector itself. The vector is filled with current features of the patch from the frame.

4.5.2 Training and testing data

Extracting HOG features from images

Training and testing images have to transformed into the features corresponding with *svm_light* library. A *Matlab* script **mainHOG.m** was created for this purpose. The scripts loads each image, computes HOG features and saves them into the text file. The script has programmable percentage of testing/training feature ratio.

Training the model

The produced training examples are used to create model. The model is created using *svm_learn* binary.

4.6 Addition information about implementation

4.6.1 Data structures

Defined in Constants.h header

EncoderCallbackData_t. Structure containing information needed by the Encoder component.

- encoder_pool
    - Pointer to the *MMAL_POOL_T* used by the encoder callback
- file
    - Pointer to the opened file to which the encoded video is saved.

CmdParameters_t. Structure containing information about command-line parameters.

- sleepTime
    - Value of type *uint32_t* defining duration (in milliseconds) of the video capture.
- detectComponent
    - Value of type *uint32_t* defining the current mode (*Render* or *Encoder*)

Defined in HOG.h header

limits_all_t. Structure containing information about boundaries of more cells or patches. See figure 4.7 to see visualization of the structure parameters.

- min_w
- max_w
- size_w
    - Number of elements in arrays min_w and max_w
- min_h
- max_h
- size_h
    - Number of elements in arrays min_h and max_h

limits_simple_t. Structure containing information about frame boundaries for each thread. See figure 4.7 to see visualization of the structure parameters.

- min_w
- max_w
- min_h
- max_h

Since the frame is saved in a memory array the indexes of the corners are computed as:

$$top_{left} = frame_{width} \cdot min\_h + min\_w \tag{4.1}$$
$$top_{right} = frame_{width} \cdot min\_h + max\_w \tag{4.2}$$
$$bottom_{left} = frame_{width} \cdot max\_h + min\_w \tag{4.3}$$
$$bottom_{right} = frame_{width} \cdot max\_h + max\_w \tag{4.4}$$

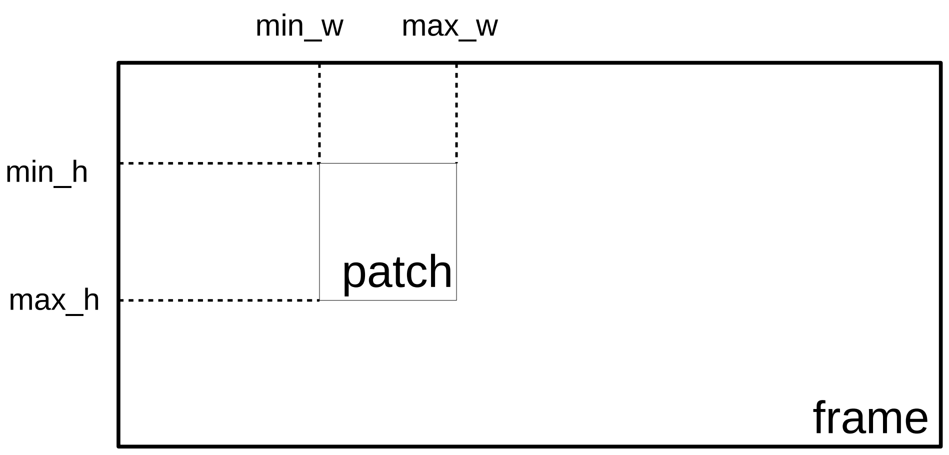


Figure 4.7: Indexing of the patches and cells limits

detection_data_t. Structure containing information about object detection.

- detected_size
  - Number of the elements in the arrays (number of patches processed by the thread)
- detected_now
  - Array of boolean values for each thread defining whether the patches contain an object
- detected_total
  - Array of type *uint32_t* values defining total number of detected objects during lifetime of the whole process for each thread
- detected_max

    - Array of type *double* values defining maximum value returned by function *classify_example_linear()* during lifetime of the whole process. See section 4.5.1 for the information about aforementioned function.
- detected_now_all
    - Pointer to array of type *uint8_t* containing boolean value of all the patches. The values represent information whether the patches contain the object.
- detected_accumulated_all
    - Pointer to array of type *uint8_t* containing *accumulated values* of all the patches. The values are used when highlighting the detected objects. See section 4.2.2 and flow chart diagram 4.6 for more information.
- patches_in_width
    - Number of the patches in the horizontal dimension. The value is used to compute the indexes of the current frame.

visualize_data_t. Structure containing information about position of each patch in the frame needed for highlighting the detected object.

- detected_size
    - Total number of patches in the frame.
- detected_w
    - Total number of patches in horizontal direction
- detected_h
    - Total number of patches in vertical direction
- mem_size_w
    - Size of the memory used by function *memcpy()* to highlight the tracking box
- video_w
    - Width of the video frame
- size_h
    - Height of the tracking box
- index_h1w1
    - Position of the top-left corner of the tracking box
- index_h1w2
    - Position of the top-right corner of the tracking box
- index_h2w1
    - Position of the bottom-left corner of the tracking box

thread_data_t. Structure containing data provided to each thread

- limits
    - Structure limits_simple_t
- all_patches_limits

  - Boundaries of all the patches processed by the thread (structure limits_all_t)
- det_data
  - Information about the object detection (structure detection_data_t)
- img
  - Pointer to array of type *uint8_t* containing current frame
- frame_histograms
  - Pointer to array of type *float* containing computed histograms for the current frame
- frame_features
  - Pointer to array of type *float* containing computed features for the current frame
- thread_control
  - Boolean value defining information if the thread should continue or cleanup and terminate
- thread_init_success
  - Boolean value defining whether the initialization of the thread was successful
- m_done
  - Semaphore (type *sem_t*) used by the thread to inform main thread the it is done with the part of computation (see flowchart diagram 4.4 for more information)
- m_go
  - Semaphore (type *sem_t*) used to notify the thread that it can continue with the computation (see flowchart diagram 4.4 for more information)
- model
  - Pointer to model (structure *MODEL* from the *svm_light* library, see section 4.5.1 for more information) loaded by the HOG component

4.6.2 Specifications of the implementation

Object of interest selection

The implementation was configured to detect vehicles in traffic.

Dataset of vehicles

The *Support Vector Machine* is trained from the *GTI vehicle image database* [4]. The database contains about 4000 vehicle and 4000 non-vehicle (images of roads, billboards, signs etc.) images. The vehicle images are divided into four groups:

- Images further away from the vehicle
- Images taken closer from the vehicle
- Images taken from the right side corder
- Images taken from the left side corner

Each image is has size 64×64 pixel and contains various vehicle types and brands.

Implementation details to allow processing at 60 frames per second

Selection of the video resolution. The camera hardware is capable of producing 60 frame per second (FPS) at maximum of 1280×720 pixel resolution. Unfortunately at this resolution the processor is not capable to maintain the frame rate. It is capable to run at 60 fps when the resolution is set to 640×320 pixels.

Overlapping of the patches. The patches are created from the shared memory buffer, which contains HOG features of the whole frame. Each patch of the size 64×64 pixel is represented by 7×7 arrays with length 36 (2×2 normalized histograms with 9 bins). The normalized histograms are shifted by the size of the cell (8×8 pixels), therefore the minimum step of the shift (patch overlapping) is $\frac{[64,\ 64]}{[8,\ 8]} = [8,\ 8]$ pixels. Using the smaller patch shifting (larger overlapping) achieves more continuous tracking, but also increases the computation cost. At the video resolution 640×320 the processor is capable to maintain the smallest possible patch shifting.

Detecting only relevant part of the frame. Since the vehicles are not expected in the every part of the frame some of the areas do not need to be scanned for vehicle objects. Half of the vertical part and some portions of horizontal part can be eliminated for this use case.

Directories hierarchy

The files provided in attachments (enclosed CD) have to following hierarchy:

- *matlab*
  - HOG
    - *Matlab* script that extracts the HOG features from the training images and inputs them to the text file in the format expected by *svm_light* library (see section 4.5.1 for detailed description).
  - SVM
    - *Matlab* script that was used for creating figures in section *Support Vector Machine* 3.2.2
  - FeaturesExtractionTesting
    - *Matlab* script that was used for testing features extraction. The results from the *C* code implementation were compared with the *Matlab* results.
- HOGdetection
  - inc
    - Directory containing header files.
  - src
    - Directory containing source files
  - Makefile
    - Used for building the project
- FeaturesExtractionTestingC
  - main.c
    - Source file used for testing features extraction. The results are compared with the *Matlab*.
- Results
  - Text file with link to website containing the testing videos.

Chapter 5

Testing of the detection

The application was tested in four following operational scenarios:

- Slower moving vehicles detected from a sidewalk of the road
- Faster moving vehicles detected from a sidewalk of the road
- Slowing down vehicles - moving forward to a road crossing
- Tracking of the vehicle from the car windshield

In all the aforementioned cases the application was able to detect the vehicles and highlight them with the tracking boxes. See the figures 5.1 and 5.2 showing the example frames of the detected vehicles. The performance of the detection can be improved by choosing exact placement of the *camera hardware*. The advantages of placement knowledge in advance are:

- The *SVM* classifier can be trained only for the expected vehicle rotation and positions, which increases detection precision.
- The precise part of the frame can be chosen for detecting to reduce computational cost of the image features.

The captured *MP4* files are available at the following http shared folder. The link is also provided in enclosed CD (see section 4.6.2).

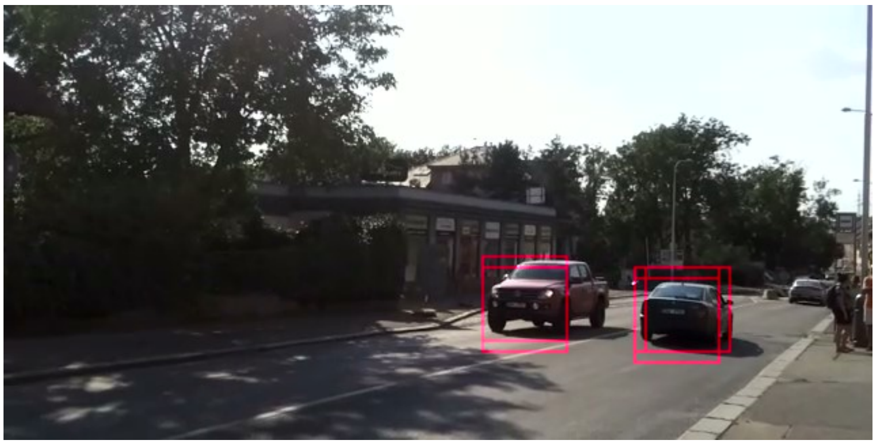


Figure 5.1: Example of the video frame, while detecting vehicles 1

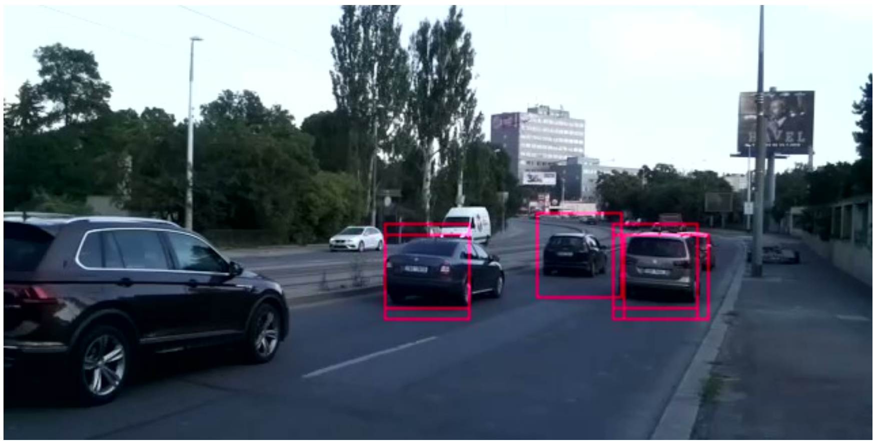


Figure 5.2: Example of the video frame, while detecting vehicles 2

Chapter 6

Conclusions

The purpose of this thesis was to implement object detection for the platform Raspberry-Pi using algorithm histogram of oriented gradients. In the introductory we got acquainted with all the necessary properties needed for the successful completion of the goal. We explained algorithm *histogram of oriented gradients* which extracts features from an image based on the direction of edges. In addition we described what does the classification algorithm linear *support vector machine*, which builds model from the training images. The trained model is then used to classify the current incoming frames. At last but not least we showed how to use *Multi-Media Abstraction Layer API* (MMAL API for short) developed by *Broadcom* to create processing pipeline. Based on the gained theoretical background we successfully developed processing pipeline for tracking vehicles, that runs on Raspberry-Pi platform with 60 frames per second at real-time. Since the used APIs were provided in *C* programming language, it was also selected for the implementation. In order to assemble the pipeline, we created custom MMAL API component that processes incoming video frames from the camera hardware and detects the present vehicles. To utilize all the cores of on-board processor the created component works with threads and synchronizes them for successful coordination. To ensure that the *C* implementation used in the processing pipeline produces expected features, their extraction was also implemented in *Matlab*. The same *Matlab* code was used when producing feature vectors from the training images. The final application was tested in real operation and empirically optimized. The provided solution is functional and satisfies the input requirements. All the created functions were documented in doxygen format and the implementation was described in detail in the chapter 4.

Bibliography

[1] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2005.

[2] RASPBERRY PI FOUNDATION. Raspberry pi foundation. `https://www.raspberrypi.org/`, April 2020. Accessed on 2020-25-4.

[3] RASPBERRY PI FOUNDATION. Raspberry pi hardware. `https://www.raspberrypi.org/documentation/hardware/raspberrypi/README.md`, April 2020. Accessed on 2020-25-4.

[4] M. Nieto J. Arróspide, L. Salgado. Vehicle image database gti. `http://www.gti.ssr.upm.es/data/Vehicle_database.html`, January 2012. Accessed on 2020-12-5.

[5] Thorsten Joachims. Svm-light support vector machine. `http://svmlight.joachims.org/`, August 2008. Accessed on 2020-12-5.

[6] Dave Jones. picamera. `https://picamera.readthedocs.io/en/release-1.13/index.html`, 2013 - 2016. Accessed on 2020-12-5.

[7] Creative Commons Attribution-ShareAlike 3.0 Unported License. Raspberry pi videocore apis. `https://elinux.org/Raspberry_Pi_VideoCore_APIs`, October 2015. Accessed on 2020-12-5.

[8] Broadcom Europe Ltd. Multi-media abstraction layer (mmal). draft version 0.1. `http://www.jvcref.com/files/PI/documentation/html/index.html`, 2012. Accessed on 2020-12-5.

[9] Satya Mallick. Histogram of oriented gradients. `https://www.learnopencv.com/histogram-of-oriented-gradients/`, December 2016. Accessed on 2020-12-7.

[10] Inc. The MathWorks. quadprog. `https://www.mathworks.com/help/optim/ug/quadprog.html`, 2020. Accessed on 2020-14-7.

[11] Inc. Wikimedia Foundation. Support-vector machine. `https://en.wikipedia.org/wiki/Support-vector_machine`, May 2020. Accessed on 2020-12-5.

[12] the free encyclopedias Wikipedia. Rolling shutter. `https://en.wikipedia.org/wiki/Rolling_shutter`, December 2019. Accessed on 2020-12-5.

Appendix A

CD Contents

- Implemented *C* and *Matlab* Code (see the section 4.6.2 about the directories hierarchy)

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Kršek** Jméno: **Ladislav** Osobní číslo: **439577**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra radioelektroniky**

Studijní program: **Otevřené elektronické systémy**

Studijní obor: **Komunikace a zpracování signálu**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Detekce objektu ve videosignálu

Název diplomové práce anglicky:

Object Detection in Video Signal

Pokyny pro vypracování:

Seznamte se s metodou histogramu orientovaných gradientů (HOG) pro detekci objektu ve videosignálu. Tuto metodu implementujte pro běh v reálném čase na platformě Raspberry Pi. Uvažujte rychlost zpracování odpovídající 60 snímkům za sekundu.

Seznam doporučené literatury:

[1] Dalal, N. and B. Triggs, 'Histograms of Oriented Gradients for Human Detection', IEEE Computer Society Conference on Computer Vision and Pattern Recognition, Vol. 1 (June 2005), pp. 886–893.
[2] Teach, Learn, and Make with Raspberry Pi – Raspberry Pi [online]. Raspberry Pi Foundation [cit. 2019-09-16]. Dostupné z: https://www.raspberrypi.org/

Jméno a pracoviště vedoucí(ho) diplomové práce:

prof. Ing. Pavel Zahradník, CSc., katedra telekomunikační techniky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **16.09.2019** Termín odevzdání diplomové práce: ____________

Platnost zadání diplomové práce: **19.02.2021**

prof. Ing. Pavel Zahradník, CSc.	doc. Ing. Josef Dobeš, CSc.	prof. Mgr. Petr Páta, Ph.D.
podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.
Datum převzetí zadání Podpis studenta