TALLINN UNIVERSITY OF TECHNOLOGY

Faculty of Information Technology

Department of Computer Control

ISS40LT

Carl-Martin Ivask 120712

Raspberry Pi based System for Visual Object Detection and Tracking

Bachelor's Thesis

Supervisors: Eduard Petlenkov, PhD

Aleksei Tepljakov, MSc

Tallinn 2015

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

I hereby declare that this Bachelor's thesis, my original investigation and achievement, submitted for the Bachelor's degree at Tallinn University of Technology, has not been submitted for any degree or examination.

Carl-Martin Ivask

-- --

(*date*) (*signature*)

Annotatsioon

Käsitletava töö eesmärk on uurida erinevaid meetodeid arvutiga reaalse maailma visuaalselt tõlgendamiseks, tutvuda OpenCV API pakutavate lahendustega ning neid implementeerida Raspberry Pi platvormile loodud objekte tuvastavas ning jälgivas rakenduses. Põhiline rõhk on praktilisel osal.

Lõputöö tulemuseks on GNU/Linux operatsioonisüsteemile loodud C/C++ programm, mis on võimeline tuvastama kasutaja poolt spetsifitseeritud parameetrite järgi kaamerapildil olevaid värvilisi objekte ning edastama nende kohta kasulikku informatsiooni (koordinaadid, pindalad, värv) kasutades sobivat andmeedastusprotokolli. Programmi lähtekood on dokumenteeritud ning loob võimaluse tarkvara efektiivseks edasiarendamiseks.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 34 leheküljel, 6 peatükki, 13 joonist, 4 tabelit.

Abstract

The aim of this thesis is to explore different methods for helping computers interpret the real world visually, investigate solutions to those methods offered by the open-sourced computer vision library, OpenCV, and implement some of these in a Raspberry Pi based application for detecting and keeping track of objects. The main focus rests on the practical side of the project.

The result of this thesis is a GNU/Linux based C/C++ application that is able to detect and keep track of objects by reading the pixel values of frames captured by the Raspberry Pi camera module. The application also transmits some useful information, such as coordinates and size, to other computers on the network that send an appropriate query. The source code of the program is documented and can be developed further.

The thesis is in English and contains 34 pages of text, 6 chapters, 13 figures, 4 tables.

Nomenclature

API	*Application programming interface*
OpenCV	*Open-sourced Computer Vision library*
UDP	*User Datagram Protocol*
IDE	*Integrated development environment*
USB	*Universal data bus*
SoC	*System-on-a-chip*
GPU	*Graphics processing unit*
CSI	*Camera Serial Interface*
blob	*A region of coloured pixels – white areas on a binary image*
object	*For this thesis, an instance of a C structure containing an object's properties*
HSV	*Hue-Saturation-Value colour space (as opposed to Red-Green-Blue, for example)*
GUI	*Graphical user interface*

Table of contents

1. Introduction ... 7
1.1 Background ... 7
1.2 Aim of thesis ... 8
1.3 Thesis outline ... 8
2. Raspberry Pi ... 9
2.1 Setting up the Raspberry Pi for work ... 10
3. OpenCV introduction ... 12
3.1 HSV colour space ... 14
4. Development of the application ... 15
4.1 Preparing captured image for object detection ... 16
4.2 Finding objects on thresholded image ... 17
4.3 Handling and tracking objects ... 19
4.4 Transmitting useful information via network ... 23
5. Using the application on Raspberry Pi ... 26
6. Ideas for further development ... 29
Conclusions ... 31
Kokkuvõte ... 32
Bibliography ... 33
Appendix 1 ... 35

1. Introduction

1.1 Background

Computer vision and visual detection of features and colours has been always been an area of interest for large companies able to invest in expensive high-end technology and machinery, but with the advent of cheap, mass-produced credit-card sized computers such as the Raspberry Pi.

The Pi microcoputer was created for the sole purpose of commercial and academical use and anyone in the world with enough enthusiasm and eagerness to learn can develop software and even whole embedded real-time systems, that with modern hardware specifications is very capable of image processing.

The applications of any sort of feature detection software or computer vision in general are numerous. To provide a few examples: guest tracking at shopping centres, surveillance systems with facial recognition, monitoring any sort of equipment, helping robots navigate and pick up objects, inspecting labels on products in factories, deep learning projects and a lot more.

The motivation behind this Bachelor’s Thesis is to form a foundation of knowledge regarding embedded systems, the basics of real-time image processing and visual detection of objects (or other features), upon which it is possible to add deeper layers of complexity and conduct further research with which to formulate projects ranging from inexpensive and small real-time systems for simple object detection or facial recognition to more complex systems such as self-learning robots and intelligent machines with the capability of visual feedback.

1.2 Aim of thesis

The primary aim is to write a C/C++ application, with the help of OpenCV libraries, which can be used on an embedded UNIX-based system, specifically a Raspberry Pi to:

1) Detect coloured blobs from live camera feed,
2) Assuming these blobs are objects (certain criteria are met), store them in memory to keep track of their coordinates and other properties,
3) Attempt to ensure realistic and linear movement of these objects,
4) Send useful information about these objects is continuously as datagrams (using network sockets bound with the UDP protocol) to any software making information requests.

1.3 Thesis outline

In Chapter 2 the reader is provided a description of the used platform, Raspberry Pi, and general guidelines for setting it up for work.

In Chapter 3 the OpenCV library and the HSV colour space are introduced, while the latter is explained in more detail, as it is the foundation on which colour-based object detection is built.

Chapter 4 describes the workflow of the created application and explains why and how OpenCV and C++ solutions were used.

Chapter 5 provides information on how the application can be used on a Raspberry Pi, as well as explains how and why command line arguments or the developed application's control panel should be used.

In Chapter 6 some ideas for further research and improvement are discussed, such as object handling and tracking algorithms, GPU acceleration, user interface features.

2. Raspberry Pi

Taking into account the relatively high performance requirements of image processing in general and the equipment currently available to the faculty, as a relatively inexpensive and powerful embedded platform the Raspberry Pi was an obvious choice. The hardware specifications taken into consideration for this work can be seen in Table 1.

Table 1. RPi hardware specifications [1], [2]

SoC	Broadcom BCM2835
CPU	700MHz ARM11 ARM1176JZF-S core. **Can be overclocked safely.**
GPU	Broadcom VideoCore IV, OpenGL ES 2.0, OpenVG 1080p30 H.264 high-profile encode/decode
Memory (SDRAM)	512Mib
USB 2.0 ports	2 (via integrated USB hub)
Video outputs	Composite RCA, HDMI (cannot be used simultaneously)
Video input	CSI
Audio outputs	TRS connector / 3.5mm jack, HDMI
Onboard storage	SD / MMC / SDIO card slot
Onboard network	10/100 wired Ethernet RJ45
Low-level peripherals	26 GPIO pins, SPI, I^2C, I^2S, UART
Power ratings / source	700 mA, (3.5 W) / 5V (DC) via Micro USB type B or GPIO
Size / weight	85.0 x 56.0 x 17 [mm] / 40g

Another contributing factor to this choice was the availability of the Pi camera module, which can capture high-definition video as well as stills and requires the user to simply update Raspberry's firmware to the latest version. It can easily be used in OpenCV based applications. Although using the officially supported camera module, which can be accessed through the MMAL and V4L APIs and is supported by numerous third-party libraries, any USB web-camera can be used.

Figure 1. RPi model B and CSI camera module

2.1 Setting up the Raspberry Pi for work

The choice in how to proceed with working with a Raspberry Pi is up to the developer, although it should be mentioned that while remote access is fairly convenient to use and simple to set up with a wireless internet USB dongle, image processing applications, even displaying live camera feed, require a lot of CPU processing power and therefore it is best if some of it can be spared by simply connecting the Pi to a monitor and wireless mouse and/or keyboard.

On the other hand, after remote access is properly configured and working, it can be used to install various updates, upgrades and most definitely OpenCV, which takes up to 10 hours to *make* (install) after downloading and unpacking and is best left to itself overnight [3], [4].

While coding can be done in numerous ways and it is entirely up to the programmer to decide which IDE and compiler should be used; a good way to save time is to write code on a personal computer and compile it on the Raspberry, because while Raspberry is pretty powerful for such a small computer, it is much more convenient to write code on a smoothly operating machine.

Overclocking

The easiest way to overclock the Raspberry Pi model B is to do it via Raspberry's configuration interface, which appears on every start-up, or can be opened using the command *sudo raspi-config*, which brings up a menu for various configuration possibilities as seen in Figure 2.

```
Setup Options

 1 Expand Filesystem             Ensures that all of the SD card storage is available to the OS
 2 Change User Password          Change password for the default user (pi)
 3 Enable Boot to Desktop/Scratch Choose whether to boot into a desktop environment, Scratch, or the command-line
 4 Internationalisation Options  Set up language and regional settings to match your location
 5 Enable Camera                 Enable this Pi to work with the Raspberry Pi Camera
 6 Add to Rastrack               Add this Pi to the online Raspberry Pi Map (Rastrack)
 7 Overclock                     Configure overclocking for your Pi
 8 Advanced Options              Configure advanced settings
 9 About raspi-config            Information about this configuration tool


                      <Select>                                  <Finish>
```

Figure 2. *sudo raspi-config*

Overclocking is recommended since image processing operations consume fairly large amounts of CPU power (if not optimized to harness the GPU instead) and if the Raspberry's airflow is above minimal (heat disperses easily), it will not damage the SoC.

The only problems that may arise are unstability when remotely accessing the Pi on which a graphical user interface server is running. In that case, the Raspberry quickly shuts down all access via network and can even freeze completely.

This means that a less powerful overclocking preset should be preferred to ensure stable execution of any application relying heavily on the CPU.

```
Choose overclock preset

None   700MHz ARM, 250MHz core, 400MHz SDRAM, 0 overvolt
Modest 800MHz ARM, 250MHz core, 400MHz SDRAM, 0 overvolt
Medium 900MHz ARM, 250MHz core, 450MHz SDRAM, 2 overvolt
High   950MHz ARM, 250MHz core, 450MHz SDRAM, 6 overvolt
Turbo  1000MHz ARM, 500MHz core, 600MHz SDRAM, 6 overvolt
Pi2    1000MHz ARM, 500MHz core, 500MHz SDRAM, 2 overvolt
```

Figure 3. Raspi-config overclocking presets

All presets up to High were tried using TightVNC as a GUI server for remote access through Windows, so stability problems may have arisen from this (TightVNC also consumes a lot of CPU power since it draws the whole desktop and every open window), but the most satisfying workflow was achieved with a simple 100MHz boost (Figure 3), while anything higher than that caused freezes and disconnections too often, or simply did not provide a tangible improvement to performance.

3. OpenCV introduction

The application written for this thesis relies heavily on computer vision, image processing and pixel manipulation, for which there exists an open source library named OpenCV (Open Source Computer Vision Library), consisting of more than 2500 optimized algorithms. Uses range from facial recognition, object identifying, classifications of human actions in videos, achieved with filters, edge mapping, image transformations, detailed feature analysis and more (Figure 4). Having Linux support, this is the perfect choice for developing an application specifically for a Raspberry Pi based system. Another positive aspect of this library is that it's written natively in C++ and therefore can be very smoothly implemented in a C/C++ application [5].

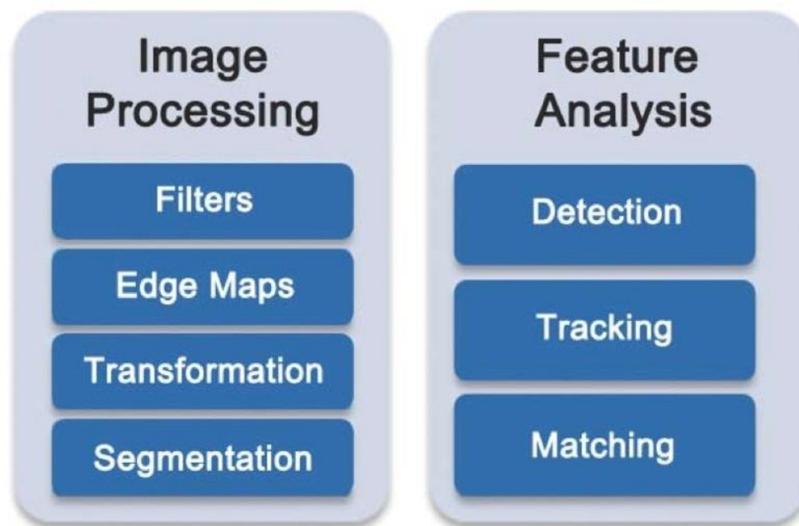

Figure 4. Partial overview of what OpenCV includes [6]

While there are numerous methods and algorithms contained within OpenCV, the most important benefit of this library for the purposes of this thesis are its basic data structures like *Mat*, which can be used to store pixel values of an image in an *n*-dimensional array, *Scalar* and *Point*, which respectively contain pixel values and coordinates of up to 3 dimensions [7].

The functions provided by this library are also necessary in the development process of the object tracking application. There are numerous options, but following the scope of this thesis, the focus is set on grabbing frames from a live camera feed [8], image thresholding using HSV colour space ranges, finding blobs and using their detected contours in a binary image and, in case a graphical user interface is enabled, displaying of image frames and a control panel for changing parameters during run-time.

Table 2. OpenCV functions most relevant to colour-based object detection, used in the code of the C++ application created for the thesis.

Function name	Library	Short description
inRange	core	Checks if an image's pixel array elements lie between the elements of two other arrays (lower and upper boundaries) and thresholds them accordingly to either black or white [10]. Usage in code seen in Listing 3 of Appendix 1.
Circle	core	Draws a circle. This function needs a destination image and the drawn circle is described with a coordinate marking the centerpoint, a radius value, colour of the circle and the thickness of its outline.
cvCreateTrackbar	highgui	Creates a trackbar (a slider) which can be used to control parameters during run-time. Can be used instead of buttons, because the latter have not yet been implemented.
imshow	highgui	Displays an image frame.
cvtColor	imgproc	Convert an image frame (pixel array) into another format. In this project it is used to convert each frame grabbed from the camera feed from BGR to HSV [9].
GaussianBlur	imgproc	Blurs an image using a Gaussian filter.
moments	imgproc	Calculates all of the moments up to the third order of a polygon or rasterized shape [16].
findContours	imgproc	Finds contours of all possible areas in a binary image and stores them in an allocated vector of points (the outermost pixels of white areas). Usage of function is seen in Listing 4 of Appendix 1.
contourArea	imgproc	Takes in a vector of points forming a contour for a region and calculates it's area value (amount of pixels).
erode	imgproc	Used on a binary image this function enlarges the area covered by black pixels, effectively removing most noise (random single white pixels) from the image [11].
dilate	imgproc	Used on a binary image after *erode* to enlarge white pixel areas, making any (relatively) large enough areas to return to their original size.

3.1 HSV colour space

The HSV colour space consists of three different descriptors, which is always the minimum number to classify a colour, first of which is *hue* that describes a colour that the human eye can see. The second is *saturation,* which describes the purity of a colour or, in computer vision, how much that respective colour is mixed with white and third is *value* (or luminosity, brightness)*,* which on the contrary represents the magnitude of black in said colour. Therefore if a pixel's saturation is high, it looks more rich, while a low saturation looks dull. A colour with a low *value* appear darker or simply black [12], [13].

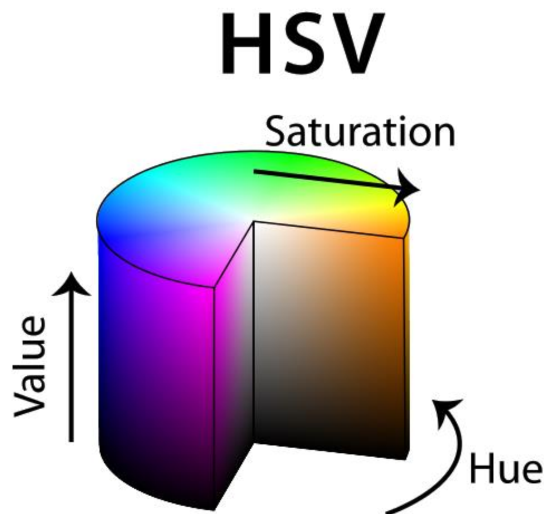

Figure 5. HSV colour cylinder [14]

The reason for using the HSV pixel format here instead of RGB, is because the HSV colour space separates color information (*saturation*) from light intensity (*value*). When ignoring *value* in the thresholding parameters, it is still possible to detect any colour from an image, given the right *hue* and *saturation* values. Another important characteristic is the circular progression (Figure 5) of *hue* values for different colours, which means that the highest *hue* value (179 in OpenCV to fit into an *unsigned char*) is simultaneously the lowest, because the space is circular not linear, and should be kept in mind when using image thresholding algorithms in an OpenCV-based application.

4. Development of the application

Capturing video with the camera module

Capturing video straight from the Raspberry Pi camera module is simple with OpenCV and at this point does not require any additional drivers or other software.

Videocapture is simply initialized by calling the *VideoCapture* class and assigning it a name. The class copy can then be used to grab frames from the camera at the start of each loop, given that the loop is executed multiple times (it is an endless loop). After initializing the camera, it is recommended to check whether or not it actually succeeded. This can be done with a simple IF statement, because the defined *VideoCapture* instance will have a function *isOpened*, which returns *true* if the camera is running. Video capturing settings can be changed within code.

When writing a program that displays any graphical items, or more specifically, displays any constantly changing images, the endless loop must include a delay to save enough time to process any draw requests (e.g. displaying an image) and for this OpenCV has the *waitKey* function, which also checks if the user pressed the key specified (Listing 1 of Appendix 1) [15].

Creating a control panel

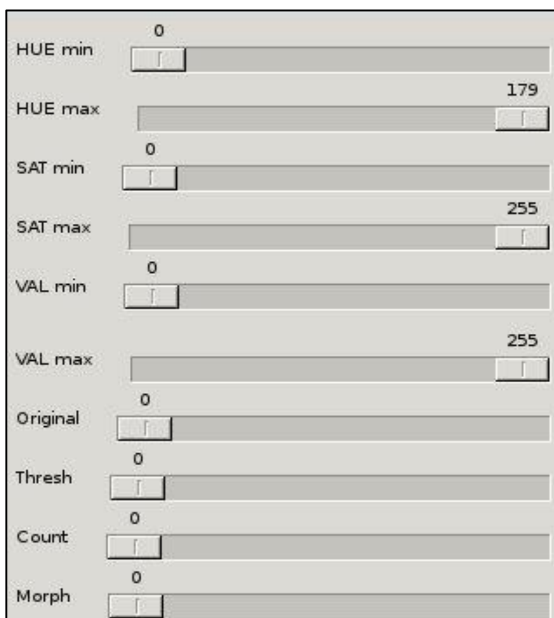

Figure 6. Example of a control panel created using cvCreateTrackbar

Creating a program that deals with image processing and not including at least something like a graphical user interface is nonsensical, because the environments, in which this sort of system works, can change a lot during run-time. For example, when not using it inside a room with constant illumination (outside), parameters first entered by the user may become obsolete since computers are a lot more sensitive to pixel values than human eyes.

Therefore a convenient feature of OpenCV would be its built-in trackbars (Table 2), which can be added to a control panel window and used to change various values during run-time (see Listing 2 of Appendix 1).

4.1 Preparing captured image for object detection

The first step towards making the image easily readable is converting the image into the HSV colour space, which is much easier to threshold (Table 2. *cvtColor*).

The second step is thresholding, which in this case is simply iterating through a pixel array of the captured image containing HSV values and setting the values of those pixels to 0 if the their values are below a lower boundary or 255 when the values are above a provided higher boundary, effectively creating a binary image. The boundaries are specified by *scalars* containing the currently set boundaries of hue, saturation and value.

$$\begin{gathered} \boldsymbol{dst\ (I) = lowerb \leq src(I)_{hue} \leq upperb} \cap \\ \cap\ \boldsymbol{lowerb \leq src(I)_{sat} \leq upperb} \cap \\ \cap\ \boldsymbol{lowerb \leq src(I)_{val} \leq upperb,} \end{gathered}$$

where **dst** is the current pixel on the destination frame (where the binary image will be stored) which will store the result of this equation (1 for white or 0 for black), **lowerb** and **upperb** are *scalars* containing set HSV ranges and **src** is the current pixel on the image frame grabbed from camera [10].

This binary image can then be read and analyzed by contour finding algorithms, thus allowing the program to count objects.

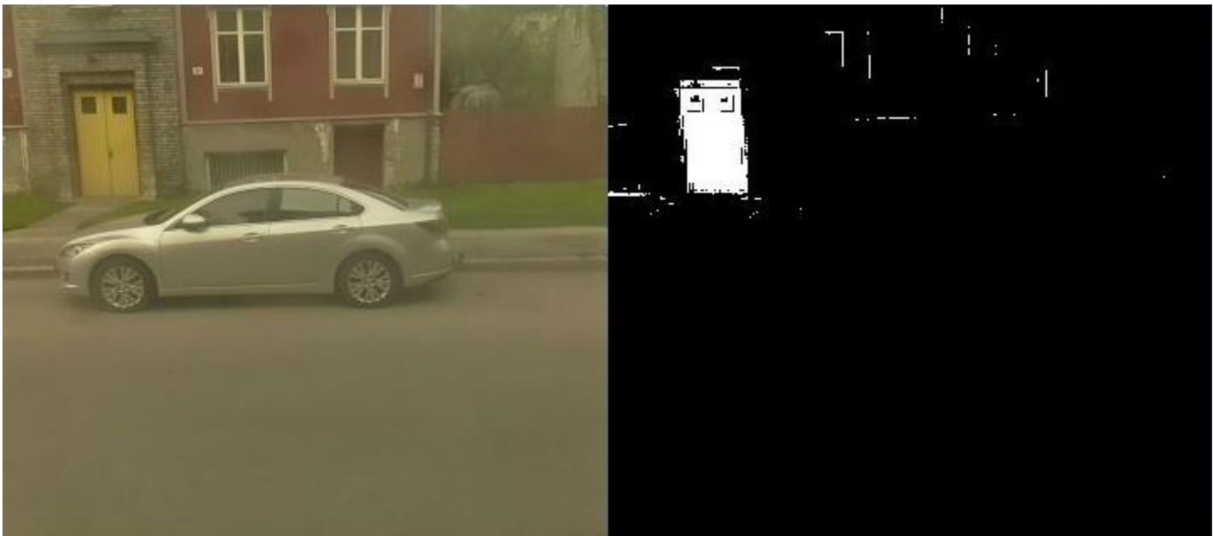

Figure 7. Image after thresholding. The selected hue range in this case was between 0 and 38, which is roughly orange or yellow (colour of door on image). The lower boundary of saturation was also raised to accommodate current lighting conditions.

The third step, erosion and dilation (Table 2), is optional because the application at hand already uses conditional checking of area values to determine whether a region is large or small enough to be considered an object. It can still be useful, when using other methods of object detection, although it does slow down the process, especially when using larger kernels.

Source code for steps 1, 2 and 3 of this section are provided in Listing 3 of Appendix 1.

Figure 8. Binary image after erode and dilate

4.2 Finding objects on thresholded image

After thresholding the captured image into a binary image, the program can begin looking for blobs – white pixel regions - and storing them in memory if they qualify as objects. For finding said regions OpenCV libraries have a function named *findContours*.

To use *findContours,* the thresholded image should first be converted into an 8-bit single-channel image (while HSV or RGB images have three 8-bit channels). While the conversion is not essential, a buffer image should be created nonetheless, to avoid any unwanted alterations, because the reference manual states that the source image will be modified [17].

Next, the program searches the image for contours, which are basically vectors of points that form a sequence of points surrounding a single colour area.

In the case of curved objects (e.g. circular), these vectors will contain large amounts of points, as to create an illusion of smooth shapes, while at the same time the function also modifies the source image so that less points need to be acquired. Controversely, when finding the contour

of a rectangular object, only four points are required (one for each corner), since a straight line can be drawn from one to another, ultimately forming a rectangle.

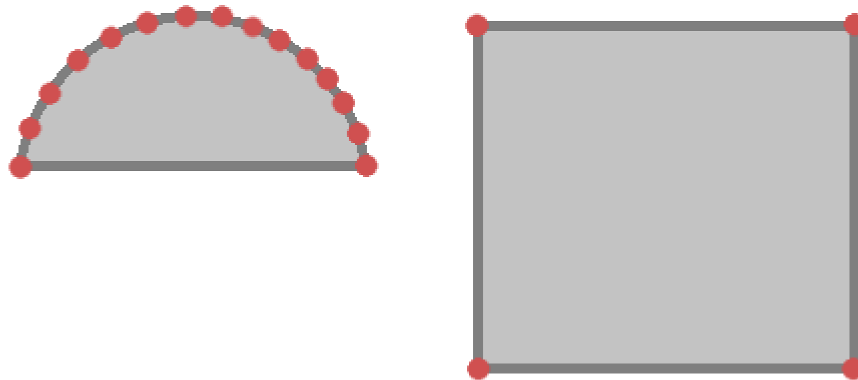

Figure 9. Illustrative contour points of a semi-circle and a strictly rectangular object

After gathering information about all the blobs on the image, the program can then calculate area size (Table 2. *contourArea*) and central coordinates (see Listing 4 of Appendix 1) for the areas within each contour.

It should be noted that these contour-bound areas are only considered objects if their area value surpasses the default or user-defined minimum values and remains below the maximum. If the user has not specified this at execution time, the default object size boundaries are calculated so that the image could ideally (side by side) fit 100 of the smallest objects or 4 of the largest objects. For example, with a default videocapture image size (256x256), the smallest objects must contain at least 656 pixels and the largest can contain up to 16,384.

While calculating the coordinates of each object, that object is also pushed to a vector of found objects, which will be used to compare the vector of existing objects and the vector of recently discovered objects.

An object has several properties (as seen in Table 3), including coordinates, area value and *removal* and *life counters*, currently tracked HSV ranges and is therefore defined as a C struct - each new object is an instance of that struct.

4.3 Handling and tracking objects

Storing objects in memory is implemented so that the application has a general idea of what describes an object in this scope, as is common in object-oriented programming. There are two possible ways this can be done: creating either a C class or a struct. Because the main difference between these two is default access levels, struct was chosen as its members' default access level is public, to avoid any unnecessary problems.

The instances of this structure are essentially the objects which the program will be handling. From the creation of these copies up to the removal or drawing of markers and transmitting information, the objects go through three phases (illustrated in Figure 10):

1) An instance of the object struct is created after finding contours and their characteristics. The created object is then pushed into the vector holding all the objects found during this cycle. It is therefore considered a **found object**.

2) The vector of found objects is then compared with another vector (holding existing objects) and any unique objects are pushed to the latter. The elements of this target vector are therefore considered to be **potentially** existing objects.

3) After counter values have been taken into account and non-existing objects are dropped, those with a high enough life value have achieved **existing** status. These are the only objects that can be marked on the original image and whose information can be transmitted via network. These objects can still be removed when their removal counters reach zero.

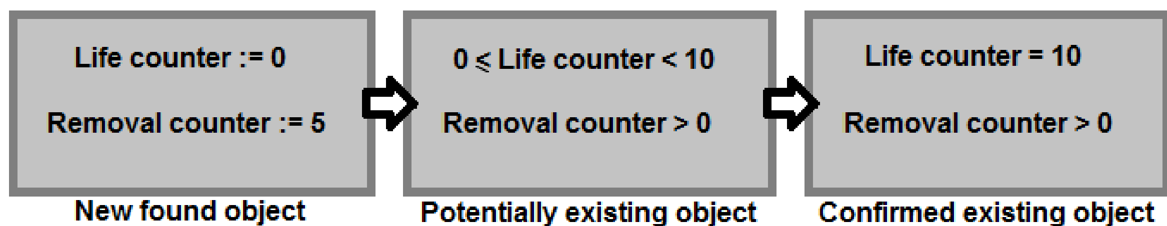

Figure 10. Three levels of object existence described using counters as criteria

Table 3. Description of the Object struct (C struct seen in Listing 12 of Appendix 1)

Member variable name	Description
Index	Stores an index given to a newly created object at the point of confirmation as an new object. Default value for all objects is 0.
X	Location of the object's centroid in relation to the X axis of the image frame.
Y	Location of the object's centroid in relation to the Y axis of the image frame.
Area	Amount of pixels in a found blob. This can be set to retain its value even when an object moves (as otherwise it will rarely remain the same).
Removal counter	Given a starting value specified by the user or a set default value (5). When this reaches zero, the object will be removed. This counter is only decreased.
Life counter	Value assignment process is the same as for the removal counter. Markers are drawn on the original image and information is transmitted via UDP only when this reaches a set value (10). This counter is only increased.
Coordinate margin	Margins for coordinate areas in which another object is compared within. This is calculated at the creation of the object using the object's Area value.
Tracked colour	This can be either an array of values or can be multiple variables containing the lower and upper boundaries of the tracked colour range.

Confirmation of found objects and placing them alongside existing objects

After objects are found and pushed into the *found objects* vector [20], they are compared to the existing objects. There are various ways of deciding if an object is indeed new or if it already exists, but in this case basic coordinate comparison is used.

If a found object's coordinates are too similar to an existing object's coordinates (defined by coordinate margins), this new object is dropped and not added to the vector of existing objects, while its coordinates are applied (if not specified otherwise by the user) to the existing one, allowing existing objects to move. This can be disabled if the user desires the objects to be more static.

The coordinate margins used in comparing objects with each other is calculated as if every object is a circle, which allows the program to easily find it's diameter, derived from its area value, and therefore the radius, which is then used as a coordinate margin within which a different object can not exist.

Source code of this function displayed in Listing 5 of Appendix 1.

Conditional checks of the existence of objects

After new objects have been added next to already existing ones, the program goes over both object vectors once more to decide which objects are (potentially) no longer existing and which are definitely still seen on the image. To do this, the program first iterates through *existing* objects again and checks if they are located inside the areas of any of the newly found objects. When this conditional check returns a positive value, the object's *life counter* is increased, as if it has existed for one more cycle. If the object at hand does not fit into the margins of any of the newly found objects' coordinates, it might not exist anymore, therefore its *removal counter* is decreased.

These aforementioned *removal* and *life* counters are used when either drawing markers where found objects are on the original image frame, or removing them during the next phase of the program, which deals with the removal of non-existing objects (those with corresponding counters at zero).

Code example shown in Listing 6 of Appendix 1.

Cleaning up after itself

After the increasing and/or decreasing of counters has been handled, it is time for the program to clean up after itself. This is simply done by iterating through the whole vector of existing objects and removing any objects that have a *removal counter* value of zero.

This is currently implemented only to save memory, but in case of a system, where all objects should be stored in memory for longer periods of time, this could prove to be a very unnecessary feature. Source code found in Listing 7 of Appendix 1.

Drawing markers on the original frame (if displayed)

The final function that takes place during the main process cycle is the drawing of markers on the original frame, if the user has not specified otherwise.

This is where *life counters* are utilized (not exclusively), as only objects with this counter's value over a specified amount (which is higher than the removal counter, by default) get their *centroids* and area-equivalent red circles drawn on the live camera feed (example shown in Figure 11). This is mainly for aesthetic purposes, but can be used to check if the user has provided correct parameters (e.g. correct HSV range).

The red circles are drawn using the function *Circle* (Table 2), which takes coordinates and a radius, and uses them to draw a circular line. The radius here is calculated using the object's stored size and treating it as if it were a perfect circle, even if in reality it's a rectangular object.

Figure 11. Drawing markers on detected objects. Function seen in Listing 8 of Appendix 1

4.4 Transmitting useful information via network

This section deals with acquiring useful information from existing objects and writing them in an easily read and parsed format to a buffer string, which is then broadcasted via a network socket using the UDP protocol. Client applications can request this information by providing a proper host name (IP address), port number and a matching *passphrase* of up to 64 bytes. Both information and passphrase datagram size limits are not pre-defined and can be chosen by the programmer.

On server side, in this case the Raspberry, using datagram communication involves:

1) **Creating a network socket with the *socket* function, which takes 3 arguments, and binding the socket to its address [18], [19].**

   The first argument is address domain of the socket, which, for the sake of possible cross-platform communication, is set to AF_INET. The other option would be AF_UNIX, which is a domain reserved for two processes that share a common file system.

   The second argument is the socket's type. Because the given application uses small messages or datagrams to send to a querying client, the *datagram socket* type is used (SOCK_DGRAM).

   The third argument is the protocol, which is left as zero, so the operating system will choose the most appropriate protocol. For datagram sockets, UDP will be used.

   An example, also used in the created program, of creating and binding a socket on Linux is seen in Listing 10 of Appendix 1 (SetupSocket).

2) **Writing useful information about existing objects into a datagram buffer.**

This part is mainly standard C++ string operations like *strncat* and *strcpy* [20] and focuses on converting useful values of existing objects like their indexes, coordinates and area into string format and appending them into the datagram buffer. The datagram also includes a timestamp (consisting of only daytime) and the total amount of objects.

Each different value is preceded by delimiters, for example in the case of the object's index, the actual value is preceded by <i>. The format or the identifiers used is entirely up to the developer, while the delimiters must include symbols that would never appear in the actual values being written into the buffer, as to make reading them less complex on the receiving side.

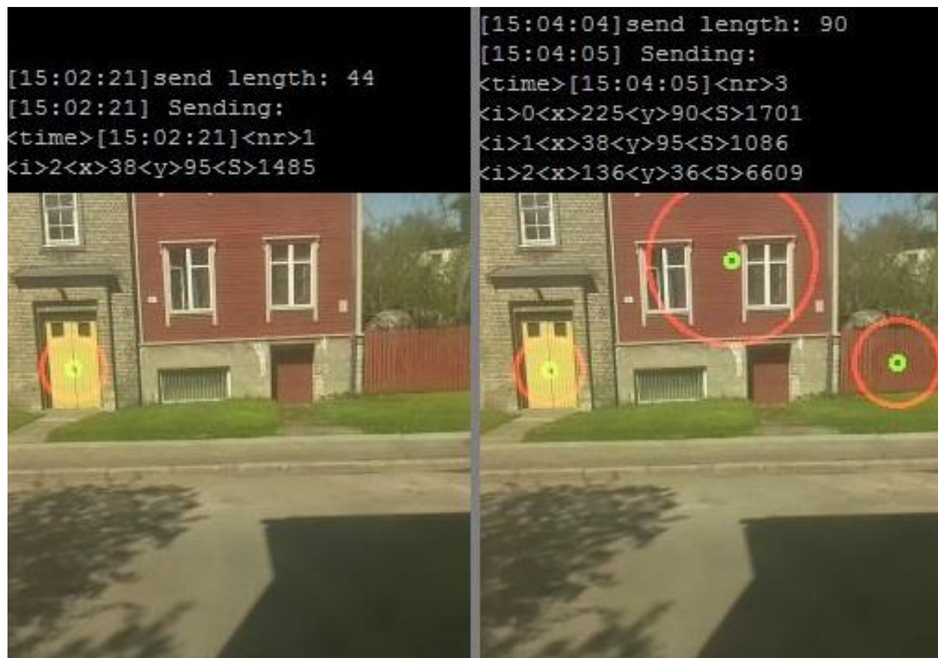

Figure 12. Examples of datagram buffer format

The source code for the function that is used to append information into a buffer variable is shown in Listing 9 of Appendix 1.

3) **Receiving and sending datagrams via the network socket [18], [19].**

Sending and receiving datagrams after the program is bound to a network socket is a simple receive-send sequence.

First, a message is received from the socket and stored in the passphrase buffer. This is the message that a client application has sent via the same socket (that they have bound to as well on their side), using the function recvfrom(). As the *flag* (fourth) argument for this function, it is recommended to specify the flag MSG_DONTWAIT, because otherwise the whole program will halt at this point, until a message is received from client software. Without that flag, the program can only properly function if there is another application constantly sending messages to this one [21]. This function also stores the sender address in memory for use in returning a message.

After receiving a correct passphrase, which is checked by comparing the received datagram and the required passphrase specified server side, the program sends back whatever is in the datagram buffer, which was filled in the previous function.

The function used for sending and receiving datagrams via network sockets in this application is presented in Listing 10 of Appendix 1 (RecvSend).

Description of simple UNIX C application for requesting information from Raspberry.

While much can be done with the information transmitted by the object detection program, for example redrawing the objects on the client machine or any number of parsing operations, a simple send-receive-print program has been written for the purpose of presenting an example (Listing 11 of Appendix 1).

5. Using the application on Raspberry Pi

The application can be run on a Linux operating system, in this case a Raspberry Pi Linux distribution (Debian Wheezy). It is preferrably run from command line and can be provided parameters by the user (currently implemented arguments described in Table 4). It is recommended the user try using the application without any parameters first to get a glimpse of how these settings change what the program is doing, especially using the colour range sliders.

Using the application

The first step for the user is to check whether or not the camera is facing whatever direction should be scanned for objects.

Next, the colour ranges must be set correctly to separate a certain colour (range) from the rest of the image. This is done via thresholding and the resulting binary image can be displayed when enabled by the user on the control panel. The contours for the white pixel regions on the thresholded image will then be found.

After the colour ranges are set and seem to be consistent enough, the colour ranges can be written down for re-use (or if such a method is implemented in the code, saved into a configuration file) and the any unnecessary windows can be disabled.

Note that trying to close continously displayed images such as the original camera feed and the thresholded image via clicking on the exit button will not work, except for the case of the control panel window, as they are redrawn each cycle, which is generally around 40-150 milliseconds.

Next, if counting objects is enabled, information will be written into the datagram buffer and red circle markers with roughly the same area value of the objects will be drawn on the original image.

In the program written for this thesis, another feature is included: debug level (seen added to the control panel in Listing 2 of Appendix 1). This is used to see what is currently being written into the datagram buffer, that will be sent to any querying software, and what the current state of objects is (all parameters of each object in the *existing objects* vector are displayed, including *life* and *removal* counters). Note that only the objects that have lived for sufficient cycles will be considered existing objects and the rest will be removed promptly.

Approximate colour ranges

Choosing a corresponding HSV range is an integral part of using the created object detection application. If the user does not know what the values on the control panel mean or how to select the desired range, a demonstrative image of hue progression can be displayed.

The drawning of such an image is provided in Listing 13 of Appendix 1, although the image drawn by this function does not currently include a scale for these values. This function could theoretically be used for creating a section of the graphical user interface, which would allow selecting the tracked range from a visible colour range instead of a slider (Figure 6).

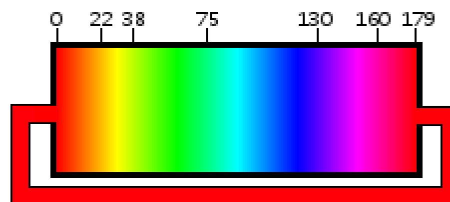

Figure 13. Hue values as a range from 0 to 179. The numbers are rough estimations as to where one colour ends and another starts, while the „pipe" represents its continuous nature, which means that if the lower boundary is set to be higher than the upper boundary on the trackbar, the range will not end at 179, but rather progresses through 0.

It should also be noted that while hue is essentially separated from saturation and value, in most cases either of the latter two also need to be to achieve satisfactory colour separation, because of imperfections in the captured image and the difference between how a computer and the human brain interpret a certain colour.

Table 4. Command line arguments, which can be passed to the program. Settings specified in this manner will override the application's default settings.

Argument	Parameters	Use result
-hue L H	L – low H – high	Set low and high boundaries for hue.
-sat L H	L – low H – high	Set low and high boundaries for saturation (colour intensity).
-val L H	L – low H – high	Set low and high boundaries for value (light intensity).
-objsize MIN MAX	MIN – minimum size MAX – maximum size	Set minimum and maximum size of objects to be considered.
-capsize H W	H – height W – width	Set height and width of captured frames in pixels.
-framesize H W	H – height W - width	Set height and width of resized images for display. This can be different from capture size and should be set smaller, because drawing windows puts a heavy load on Raspberry's CPU. Does **NOT** affect object detection.
-nogui	None	Disables graphical user interface. Only recommended for use in synergy with properly entered command-line parameters. Reduces CPU load significantly.
-morph X	X – [0..2]	Sets morphological filtering level.
-noblur	None	Disables applying gaussian blur when thresholding image. This will result in more noise in the resulting image.
-nocount	None	Disables all object detection- and counting-related features.
-udppass [string]	64 byte word	Sets the passphrase which client software must use to query information from Raspberry.
-udpport X	X – [2000.. 65535]	Sets port on which information is broadcast.
-rmstart X	X – [..]	Set starting *removal counter* to provided value.
-drawmin X	X – [..]	Set minimum lifetime required for objects to be marked on the original frame.
-debug X	X – [0..10]	Set debug level. Can be used to print information to the console, such as the contents of the string buffer sent via UDP or all objects and their *life* and *removal* counters.
-help	None	Display all possible arguments and their entering formats.

6. Ideas for further development

Multiple colour ranges

The first idea, which is not really a different method but an extenstion of what is already implemented, was to have multiple simultaneous colour ranges used to threshold multiple binary images, which would then be thrown to the contour finding algorithm. This would provide the application the capability to look for various differently coloured objects and would come in handy when trying to detect objects of colours that are not next to each other (e.g. blue and orange).

Implementing this would require the use of similar methods as seen in the current object detection and tracking functions, but could be extremely CPU-intensive, which a Raspberry Pi might not be able to handle, but could provide more flexibility to a colour-based object detection program.

GPU accelerated source code

Another possibility, which came into question during the very beginning of this project, is GPU acceleration/optimization of C++ code for OpenCV projects. This would increase the performance of image processing tasks dramatically [22], since the current framerate is abysmal with larger capture and display sizes and extremely high CPU loads (over 99%) can cause the Raspberry Pi to freeze and prevent remote access from the network, which can prevent smooth testing of the application being developed, because in that case the Pi needs to be hard-reset.

Object detection and handling algorithms

Considering the current application's object handling code, much can be added and improved, such as better algorithms for tracking objects not using only coordinate margins – the current version works most of the time, but it can be fooled by objects hiding behind each other, for example. There is a possible solution to this in the segmentation of overlapping objects, which can include the use of OpenCV *watershed* algorithm [23], [24] or various other available and well-documented methods [25], [26]. Since colour-based detection has its limits, an approach combining colours, segmentation and especially algorithms for finding corners, which are essentially areas of pixels, where there are texture or edges going in at least two separate directions [25], would be the end-goal of a fully developed, functional, maintainable and cost-effective object detection and tracking system.

User interface features

An earlier version of the current application included the functionality to click on the original image window to select a colour range instead of dragging trackbars in the control panel. This worked using an OpenCV function, *SetMouseCallback*, which read cursor actions and passed the selected coordinates to another function, which in turn iterated through a patch of pixels around the provided coordinates, averaged together the area's colour range and changed the values concerning hue, saturation and value (both lower and upper boundaries). This functionality was dropped after running into conflict with access permissions within the ColourTracking class and was deemed more of a convenience than a necessity.

Selecting the HSV range via clicking on the image and many other possibilities, such as selecting specific areas of a frame within which to count objects or at the very least displaying coordinates when the cursor is hovering over a frame, would arise if this functionality was reimplemented and/or developed further.

Another helpful tool to implement would be a HSV based colour wheel, from which the user can click and drag over certain areas to define ranges. This would be much more intuitive than adjusting sliders on a linear range and would be more descriptive of the abstract cylindrical shape of the HSV range.

Histogram based self-calibration of tracked ranges

OpenCV features functionality to calculate the histogram of a frame, which are essentially collected counts of data organized into a set of predefined bins (a sub-range of values, for example all *hues* that appear as green, or all *values* that describe dark or bright colours, even gradients and directions) [27]. This information could be used to calculate the average illumination of an image, compared to previous, different values and automatically calibrate thresholding ranges accordingly, therefore providing more adaptive and consistent results when the application is used outside (daylight varies a lot, especially regarding pixel values).

Conclusions

Identifying objects via filtering of colours (pixel HSV values) is only one of many different methods that can be used for such a system. Colour-based object detection using colours is definitely an effective method, especially when dealing with objects that generally have no constant distinguishable features or corners. Such objects can be balloons and other round coloured objects, or even spots of paint, to provide a few examples.

The main problem with using colours, or more accurately, pixel values, for this purpose is the effects of inconsistent lighting, which a computer can be very sensitive towards, while a human eye can only detect a slight difference. The main cause of this problem is computers' lack of an inherent learning ability, which prevents adapting to new conditions such as a spike in the amount of yellow and white pixels on an image due to natural (varying amounts of daylight) and/or synthetic lights (flickering).

Considering the aim of this thesis was to create a practical C/C++ application for use in a Raspberry Pi based system that detects objects depending on their colour and attempts to ensure linear and realistic movement tracking, the results are satisfactory enough, while the existing code can and should be optimized further, using a wider array of different functions available in either OpenCV or C++ standard libraries, as well as improved style-wise, towards using more consistent standards and practices of programming in C++.

The realization of communication via UDP is, while currently basic in terms of security and usage, successfully implemented in the source code. Albeit being functional, it would probably be wise to separate the module at hand from the rest of the application, so it could be used in parallel with the main program while maintaining a certain level of independence on both sides. Another improvement worth mentioning would be the controlling of the application remotely, which would require to the program to be able to discern control words from the passphrase, similarly to how command line arguments are parsed.

OpenCV and C++ both have vast libraries, with seemingly endless possibilities regarding visual operations, image processing, handling memory and much else. Therefore it is quite necessary to conduct further research in this area. While the currently developed application has been realized according to the expectations set at the beginning of this work, a wider (or more specific) understanding of the used tools is required to create flawless and profitable real-time systems that require object detection and tracking.

Kokkuvõte

Objektide identifitseerimine värvide (pikslite HSV väärtuste) järgi on kõigest üks meetoditest, mida saab taolise süsteemi puhul rakendada. Värvide järgi objektide tuvastamine on kahtlemata efektiivne meetod, eriti objektide puhul, millel pole püsivaid või eristatavaid omadusi või nurkasi. Sellised objektid võivad olla näiteks õhupallid või muud värvilised ümarad asjad, isegi värviplekid mingil pinnasel.

Põhiline probleem värvieristuse puhul antud eesmärgil on ebaühtlase valgustuse mõjud, mille suhtes võib arvuti väga tundlik olla, samal ajal kui inimestel võivad väiksemad muutused jääda kahe silma vahele. Selle probleemi üks olulisemaid põhjuseid on arvutite õppimisvõime puudumine, mistõttu ei pruugi selline süsteem suuta adapteeruda uute tingimustega, näiteks päikese või sünteetilise valguse poolt põhjustatud järsu värvide muutusega kogu pildil.

Võttes arvesse sissejuhatuses püstitatud eesmärki luua Raspberry Pi-le C/C++ keeles rakendus, mis tuvastaks värvi järgi objekte ning üritaks tagada leitud objektide lineaarne ning realistlik liikumine, võib töö tulemusega piisavalt rahul olla. Siiski on ilmne, et olemasolevat lähtekoodi on vaja edasi arendada, kasutades rohkem nii OpenCV kui ka C++ standardteekide poolt pakutavaid lahendusi ning parandades koodistiili vastavalt levinumatele programmeerimisstandarditele ning headele tavadele.

Andmete edastamise UDP protokolli teel, olles turvalisuse ning rakenduslikkuse poole pealt antud juhul algeline, on loodud rakenduses edukalt implementeeritud. Tõenäoliselt oleks mõistlik antud moodul eraldada ülejäänud programmist nii, et seda saaks kasutada paralleelselt põhirakendusega, samal ajal säilitades mõlema mooduli iseseisvust. Veel üks mainimist väärt täiustus oleks programmi kaugjuhtimise implementatsioon, mille puhul peaks rakendus suutma saadetud sõnumis eristada parooli ning kontrollsõnu (ja nende parameetreid), sarnaselt käsurealt sisestatud argumentide lugemisele.

C++ ning OpenCV on mõlemad massiivsed teegikogumikud (üks rohkem kui teine) ning sisaldavad üüratult palju võimalusi visuaalseteks operatsioonideks, pilditöötluseks, mäluhalduseks ning paljukski muuks. Antud teema vajab süvitsi uurimist, et jõuda parema arusaamiseni kasutatud tööriistadest, misläbi oleks võimalik luua efektiivsemaid ning reaalselt rakendatavaid süsteeme objektide visuaalseks reaalajas tuvastamiseks ning jälgimiseks.

Bibliography

[1] RPi Hardware. [WWW] http://elinux.org/RPi_Hardware (16.03.2015)

[2] Frequently Asked Questions. [WWW] https://www.raspberrypi.org/help/faqs/ (16.03.2015)

[3] OpenCV Installation in Linux. [WWW] http://docs.opencv.org/doc/tutorials/introduction/linux_install/linux_install.html#linux-installation (16.03.2015)

[4] Installing OpenCV on a Raspberry Pi [WWW] http://robertcastle.com/2014/02/installing-opencv-on-a-raspberry-pi/ (17.03.2015)

[5] OpenCV About. [WWW] http://opencv.org/about.html (16.03.2015)

[6] Coombs, J., Prabhu, R. OpenCV on TI's DSP+ARM platforms: Mitigating the challenges of porting OpenCV to embedded platforms. [WWW] http://www.embedded-vision.com/platinum-members/texas-instruments/embedded-vision-training/documents/pages/opencv-ti%E2%80%99s-dsparm%C2%AE-plat (21.05.2015)

[7] OpenCV API Reference : Basic Structures. [WWW] http://docs.opencv.org/modules/core/doc/basic_structures.html (18.03.2015)

[8] OpenCV API Reference : Reading and Writing Images and Video [WWW] http://docs.opencv.org/modules/highgui/doc/reading_and_writing_images_and_video.html#videocapture (19.03.2015)

[9] OpenCV API Reference : Miscellaneous Image Transformations [WWW] http://docs.opencv.org/modules/imgproc/doc/miscellaneous_transformations.html#cvtcolor (20.03.2015)

[10] OpenCV API Reference : Operations on Arrays [WWW] http://docs.opencv.org/modules/core/doc/operations_on_arrays.html#inrange (20.03.2015)

[11] OpenCV API Reference : Image Filtering [WWW] http://docs.opencv.org/modules/imgproc/doc/filtering.html#erode (21.03.2015)

[12] Hue, Saturation & Value. The Characteristics of Color [WWW] http://www.greatreality.com/color/ColorHVC.htm (24.03.2015)

[13] Color Detection & Object Tracking [WWW] http://opencv-srf.blogspot.com/2010/09/object-detection-using-color-seperation.html (16.03.2015)

[14] HSL and HSV [WWW] http://en.wikipedia.org/wiki/HSL_and_HSV (16.05.2015)

[15] OpenCV API Reference : User Interface [WWW] http://docs.opencv.org/modules/highgui/doc/user_interface.html?highlight=waitkey#waitkey (18.03.2015)

[16] OpenCV Tutorials : Making Your Own Linear Filters! [WWW] http://docs.opencv.org/doc/tutorials/imgproc/imgtrans/filter_2d/filter_2d.html?highlight=kernel (30.03.2015)

[17] OpenCV API Reference : Structural Analysis and Shape Descriptors [WWW] http://docs.opencv.org/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html#findcontours (29.03.2015)

[18] Sockets Tutorial. The client server model. [WWW] http://www.linuxhowtos.org/C_C++/socket.htm (21.04.2015)

[19] Linux Programmer's Manual [WWW] http://man7.org/linux/man-pages/man7/unix.7.html (21.04.2015)

[20] Standard C++ Library reference [WWW] http://www.cplusplus.com/reference/cstring/ (20.03.2015)

[21] Linux manual page : recvfrom [WWW] http://linux.die.net/man/2/recvfrom (22.04.2015)

[22] OpenCV CUDA acceleration [WWW] http://opencv.org/platforms/cuda.html (29.04.2015)

[23] Count and segment overlapping objects with Watershed and Distance Transform [WWW] https://opencv-code.com/tutorials/count-and-segment-overlapping-objects-with-watershed-and-distance-transform/ (29.03.2015)

[24] OpenCV API Reference : Miscellaneous Image Transformations http://docs.opencv.org/modules/imgproc/doc/miscellaneous_transformations.html?highlight=watershed#cv2.watershed (29.03.2015)

[25] Bradski, G., Kaehler A. Learning OpenCV: Computer Vision in C++ with the OpenCV Library. Sebastopol, California : O'Reilly Media, Inc., 2008.

[26] Laganière, R. OpenCV 2 : Computer Vision Application Programming Cookbook. Birmingham, UK : Packt Publishing, 2011.

[27] OpenCV Tutorials. Image Processing. Histogram Calculation [WWW] http://docs.opencv.org/doc/tutorials/imgproc/histograms/histogram_calculation/histogram_calculation.html (21.05.2015)

Appendix 1

It should be noted that while the class header is provided in the final listing, implementations have only been provided for the most important functions.

Listing 1. Main program (main.cpp)

```
#include "ColourTracking.hpp"
#include "opencv2/highgui/highgui.hpp"
#include <iostream>

using namespace std;
using namespace cv;

int main(int argc, char **argv)
{
    ColourTracking ct;

    if (ct.CmdParameters(argc, argv) < 0) return -1; /* parse command line
arguments */


    VideoCapture cap(0); /* initialize camera & video capturing */

    if (!cap.isOpened()) /* if camera failed to initialize, exit program */
    {
         cout << ct.ts() << " Problem loading the camera. Exiting..\n";
         return -1;
    }

    cap.set(CV_CAP_PROP_FRAME_WIDTH, ct.width());   /* set width and */
    cap.set(CV_CAP_PROP_FRAME_HEIGHT, ct.height()); /* height of captured
frame */

    cout << ct.ts() << "Camera frame ";   /* print current frame size */
    cout << " HEIGHT:" << cap.get(CV_CAP_PROP_FRAME_HEIGHT);
    cout << " WIDTH:" << cap.get(CV_CAP_PROP_FRAME_WIDTH) << endl;


    ct.CreateControlWindow(); /* create control panel with trackbars */

    while (true)
    {

        ct.t_start(); /* starting point for time measurement */

        bool bSuccess = cap.read(ct.imgOriginal);
        if (!bSuccess){
            cout << ct.ts() << " Problem reading from camera to Mat.\n";
            return -1;
        }
```

```
        ct.Process(); // all major functions packed into one

        ct.Display(); /* display original and/or thresholded frame */

        ct.t_end();

        if (ct.getGUI()) {
            if (waitKey(ct.delay()) == ESCAPE)
            {
                cout << ct.ts() << " ESC key pressed by user. Exiting..\n";
                return -1;
            }
        }
    }

    return 0;
}
```

Listing 2. Creating a control panel using OpenCV trackbars

```
bool ColourTracking::CreateControlWindow()
{
    if (bGUI){

        namedWindow("Control", CV_WINDOW_NORMAL);

        cvCreateTrackbar("HUE min", "Control", &iHSV[0], 179);
        cvCreateTrackbar("HUE max", "Control", &iHSV[1], 179);
        cvCreateTrackbar("SAT min", "Control", &iHSV[2], 255);
        cvCreateTrackbar("SAT max", "Control", &iHSV[3], 255);
        cvCreateTrackbar("VAL min", "Control", &iHSV[4], 255);
        cvCreateTrackbar("VAL max", "Control", &iHSV[5], 255);
        cvCreateTrackbar("Original", "Control", &iShowOriginal, 1);
        cvCreateTrackbar("Thresh", "Control", &iShowThresh, 1);
        cvCreateTrackbar("Count", "Control", &iCount, 1);
        cvCreateTrackbar("Morph", "Control", &iMorphLevel, 2);
        cvCreateTrackbar("Debug", "Control", &iDebugLevel, 3);
        cvCreateTrackbar("Move", "Control", &iObjMove, 1);
    }

    return true;
}
```

Listing 3. Thresholding the image and applying erode and dilate (morphing)

```
void ColourTracking::ThresholdImage(cv::Mat src, cv::Mat& dst, int hsv[],
bool blur)
{
    // container for HSV image
    cv::Mat buf;

    // RGB -> HSV
    cv::cvtColor(src, buf, cv::COLOR_BGR2HSV);
    if (blur) cv::GaussianBlur(buf, buf, cv::Size(5,5), 0,0);

    // HSV -> binary (black&white)
    if (hsv[0] <= hsv[1]) {
        cv::inRange(buf, cv::Scalar(hsv[0], hsv[2], hsv[4]),
cv::Scalar(hsv[1], hsv[3], hsv[5]), dst);
    }
    else {
        cv::Mat higher, lower;
        cv::inRange(buf, cv::Scalar(hsv[0], hsv[2], hsv[4]),
cv::Scalar(HHUE, hsv[3], hsv[5]), higher);
        cv::inRange(buf, cv::Scalar(LHUE, hsv[2], hsv[4]),
cv::Scalar(hsv[1], hsv[3], hsv[5]), lower);
        dst = higher + lower;
    }
}


void ColourTracking::MorphImage(unsigned int morph, int size, cv::Mat src,
cv::Mat& dst)
{
    cv::Mat buf = src; /* buffer Mat on which to use erode and dilate */

    if (morph > 0) cv::erode(buf, buf,
cv::getStructuringElement(cv::MORPH_ELLIPSE, cv::Size(size, size)));
    if (morph > 1) cv::dilate(buf, buf,
cv::getStructuringElement(cv::MORPH_ELLIPSE, cv::Size(size, size)));
    if (morph > 1) cv::dilate(buf, buf,
cv::getStructuringElement(cv::MORPH_ELLIPSE, cv::Size(size, size)));
    if (morph > 0) cv::erode(buf, buf,
cv::getStructuringElement(cv::MORPH_ELLIPSE, cv::Size(size, size)));

    dst = buf;
}
```

Listing 4. Finding contours on the thresholded image

```
int ColourTracking::FindObjects(cv::Mat src, float minsize, float maxsize,
std::vector<Object>& found)
{
    cv::Mat imgBuffer8u;

    src.convertTo(imgBuffer8u, CV_8U);

    std::vector<std::vector<cv::Point> > contours;

    cv::findContours(imgBuffer8u, contours, CV_RETR_EXTERNAL,
CV_CHAIN_APPROX_SIMPLE);

    found.clear(); // clear vector to make room for new objects

    std::vector<cv::Moments> mv; // temporary moment vector
    std::vector<float> sv; // temporary area vector
    std::vector<cv::Point> mc; // temporary mass center vector (location)


    for (unsigned int i = 0; i < contours.size(); i++) { // get moments of
objects

        sv.push_back (contourArea(contours[i]));


        if (sv.back() >= minsize && sv.back() <= maxsize) {
            mv.push_back (cv::moments(contours[i], false));
        }
        else sv.pop_back();

    }

    // get mass centers and create new objects in one loop
    for (unsigned int i = 0; i < sv.size(); i++) {

        mc.push_back (cv::Point((int) (mv[i].m10/mv[i].m00), (int)
(mv[i].m01/mv[i].m00)));

        // Object arguments: new index, x, y, area, coordinate margin,
remove counter
        found.push_back (Object(i, (int) mc[i].x, (int) mc[i].y, sv[i],
rm_default, iHSV));
    }

    // returns number of mass centers (aka objects)
    return found.size();
}
```

Listing 5. Pushing found and confirmed objects to the vector of existing objects

```
unsigned int ColourTracking::AddNewObjects(std::vector<Object> found,
std::vector<Object>& exist)
{
    bool isnew = true;

    if (!exist.empty()) {
        if (!found.empty()) {

            // add currently found objects to vecExistingObjects
            for (unsigned int i = 0; i < found.size(); i++) {

                isnew = true;

                for (unsigned int j = 0; j < exist.size(); j++) {

                    if (found[i].x >= (exist[j].x - exist[j].cm) &&
found[i].x <= (exist[j].x + exist[j].cm)) {
                        if (found[i].y >= (exist[j].y - exist[j].cm) &&
found[i].y <= (exist[j].y + exist[j].cm)) {

                            isnew = false;
                            if (iObjMove == ENABLED) {
                                exist[j].x = found[i].x;
                                exist[j].y = found[i].y;
                            }

                            break;
                        }
                    }
                }

                // is a new object, add to existing objects
                if (isnew){

                    IDcounter++;
                    found[i].index = IDcounter;

                    exist.push_back (found[i]);
                }
            }
        }
    }
    else {
        if (!found.empty()) exist = found;
    }

    // return size of vecExistingObjects
    return exist.size();
}
```

Listing 6. Increasing and decreasing removal and life counters

```
void ColourTracking::ExistentialObjects(std::vector<Object> found,
std::vector<Object>& exist)
{
    bool addrm;
    unsigned int i, j;

    // if object has not been detected for too long, start decreasing
rm_counter
    // when it reaches zero or below, the object will be deleted during
cleanup
    if (!exist.empty()){

        if (!found.empty()){
            for (i = 0; i < exist.size(); i++){

                addrm = false;

                for (j = 0; j < found.size(); j++){
                    if (FitMargin(exist[i].x, exist[i].y, found[j].x,
found[j].y, found[j].cm)) {
                        if (exist[i].lifecnt < MinLife) exist[i].lifecnt++;
                        break;
                    }

                }

                if (j == found.size() && !FitMargin(exist[i].x, exist[i].y,
found[j].x, found[j].y, found[j].cm))
                    addrm = true;

                if (addrm) exist[i].removcnt--;
            }
        }
        else {
            for (i = 0; i < exist.size(); i++){  // if no objects are found
at all
                exist[i].removcnt--;
            }
        }
    }
}
```

Listing 7. Cleanup of objects with the removal counter at zero

```
unsigned int ColourTracking::CleanupObjects(std::vector<Object>& exist)
{
    if (!exist.empty()) {
        // remove objects that no longer exist
        exist.erase(std::remove_if(exist.begin(), exist.end(),
                                [](const Object & o) { return o.removcnt == 0; } ),
exist.end());

    }

    // return size of vecExistingObjects
    return exist.size();
}
```

Listing 8. Drawing markers on displayed camera feed.

```
void ColourTracking::DrawCircles(cv::Mat src, cv::Mat& dst,
std::vector<Object> obj)
{
    dst = cv::Mat::zeros(src.size(), src.type());
    float rad;

    if (!obj.empty()) {
        for (unsigned int i=0; i<obj.size(); i++) {

            if (obj[i].lifecnt >= MinLife) {

                // circle area = pi * radius^2
                rad = sqrt(obj[i].area/PI_VALUE);
                cv::circle(dst,cv::Point(obj[i].x,obj[i].y), rad,
cv::Scalar(0,0,255), 2, 8, 0);

                if (iDebugLevel > 0)
cv::circle(dst,cv::Point(obj[i].x,obj[i].y), 3, cv::Scalar(0,255,0), 2, 8,
0); // draw mass center
            }
        }
    }

    if (imgOriginal.size() == imgCircles.size()) {
            imgOriginal = imgOriginal + imgCircles; /* add drawn circles */
    }
}
```

Listing 9. Writing object information into a buffer string

```
void ColourTracking::WriteSendBuffer(std::vector<Object> obj, char* send)
{
    unsigned int k = 0;

    for (unsigned int i = 0; i < obj.size(); i++) {

        if (obj[i].lifecnt >= MinLife) k++; // real object amount
    }

    bzero(send, 2048); /* flush send buffer */

    if (iCount != 0){
        std::string amt = std::to_string(k);
        std::string time = ts();

        strcat(send, "<time>"); /* append timestamp to sent message */
        strncat(send, time.c_str(), time.size());

        strcat(send, "<nr>");
        strncat(send, amt.c_str(), amt.size()); /* append object amount */

        strcat(send, "\n");

        for (unsigned int i = 0; i < obj.size(); i++){

            if (obj[i].lifecnt >= MinLife) {

                std::string ind = std::to_string(obj[i].index);
                std::string x = std::to_string(obj[i].x);
                std::string y = std::to_string(obj[i].y);
                std::string s = std::to_string((int) obj[i].area);

                strcat(send, "<i>");
                strncat(send, ind.c_str(), ind.size());

                strcat(send, "<x>");
                strncat(send, x.c_str(), x.size());

                strcat(send, "<y>");
                strncat(send, y.c_str(), y.size());

                strcat(send, "<S>");
                strncat(send, s.c_str(), s.size());

                strcat(send, "\n"); /* append endline for each object */
            }

        }
        strcat(send, "\0");
    }
    else strcpy(send, "<start>NOT_COUNTING<end>\n");
```

```
    if (iDebugLevel == 3){
        std::cout << ts() << " Sending:\n" << send;
        std::cout << "\n" << ts() << "send length: " << strlen(send) <<
std::endl;
    }
}
```

Listing 10. Broadcasting information via network sockets

```
void ColourTracking::SetupSocket()
{
    sockfd = socket(AF_INET, SOCK_DGRAM, COMM_PROTOCOL);

    bzero((char *) &server_addr, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(comm_port);
    bind(sockfd, (struct sockaddr *) &server_addr, sizeof(server_addr));
}


void ColourTracking::RecvSend(char* pass, char* send)
{
    bzero(pass,64); /* flush pass buffer */
    clientlen = sizeof(client_addr);

    recvfrom(sockfd, pass, 64, MSG_DONTWAIT,
    (struct sockaddr*)&client_addr, &clientlen);

    if (!strcmp(pass,comm_pass)) {
        sendto(sockfd, send, strlen(send), 0,
        (struct sockaddr *) &client_addr, sizeof(client_addr));
    }
}
```

Listing 11. Client-side C code example for requesting and receiving datagrams

```c
#include <sys/types.h>   /* These five libraries are always required */
#include <sys/socket.h>
#include <netinet/in.h> /* for example, this is required to declare */
#include <arpa/inet.h>  /* the socklen_t type variables             */
#include <netdb.h>

#include <stdio.h>
#include <string.h>

#define PASSPHRASE "[Insert passphrase here]"
#define PORT_NR [Insert port number here]
#define PASS_SIZE 64
#define MSG_SIZE 2048
#define IP_ADDRESS "[Insert Raspberry's IP address here]"

int main(int argc, char** argv)
{
      socklen_t sock, bytes_recv, sin_size;
      struct sockaddr_in server_addr;
      struct hostent *host;
      char send_data[PASS_SIZE], recv_data[MSG_SIZE];
      host = (struct hostent *) gethostbyname((char*)IP_ADDRESS);
      sock = socket(AF_INET, SOCK_DGRAM, 0);
      server_addr.sin_family = AF_INET;
      server_addr.sin_port = htons(PORT_NR);
      server_addr.sin_addr = *((struct in_addr *)host->h_addr);
      bzero(&(server_addr.sin_zero), 8);
      sin_size = sizeof(struct sockaddr);

      while (true) {
            // Request datagram from Raspberry
            sendto(sock, PASSPHRASE, PASS_SIZE, 0,
                  (struct sockaddr *)&server_addr,
                   sizeof(struct sockaddr));

            // Flush message buffer before receiving new info
            bzero(recv_data, MSG_SIZE);

            // Receive datagram from Raspberry, store into string
            bytes_recv = recvfrom(sock, recv_data, MSG_SIZE, 0,
                  (struct sockaddr *)&server_addr, &sin_size);

            // Print whatever the Raspberry is transmitting
            printf("%s\n", recv_data);
      }
}
```

Listing 12. ColourTracking class header, ColourTracking constructor, Object struct definition

```
#ifndef _ColourTracking_HPP_
#define _ColourTracking_HPP_

#define COMM_PORT 60606
#define COMM_PROTOCOL 0 // UDP
#define COMM_PASS "getobjectinfo"

#define CAP_HEIGHT 256
#define CAP_WIDTH 256
#define ENABLED 1
#define DISABLED 0
#define MAX_CYCLE_T 200
#define MIN_CYCLE_T 50

#define LHUE 0
#define LSAT 0
#define LVAL 0
#define HHUE 179
#define HSAT 255
#define HVAL 255

#define ESCAPE 27
#define DEF_INTERVAL 100
#define PI_VALUE 3.1415926535
#define DEFAULT 3
#define DEF_DEBUG 1
#define MORPH_KERNEL_SIZE 3

// approximate high hues of colours
#define ORANGE 22
#define YELLOW 38
#define GREEN 75
#define BLUE 130
#define VIOLET 160
#define RED 179

#include "opencv2/core/core.hpp"
#include <chrono>

#include <netinet/in.h>

class ColourTracking
{
    /******************** Private access variables ********************/
    /***************** (e.g. configuration parameters) ****************/
    private:

    // Image pixel arrays
    cv::Mat imgThresh;
    cv::Mat imgCircles;
```

```
    // do counting; show unaltered image; show thresholded image; GUI; blur
when thresh
    int iCount;
    int iShowOriginal;
    int iShowThresh;
    bool bGUI;
    bool bThreshBlur;

    // hue/saturation/light intensity values; morph level; debug level
    int iHSV[6];
    int iMorphLevel;
    int iDebugLevel;

    // main loop delay; captured frame height; captured frame width
    unsigned int uiDelay;
    unsigned int uiCaptureHeight;
    unsigned int uiCaptureWidth;
    unsigned int uiFrameHeight;
    unsigned int uiFrameWidth;
    bool ResizeImages;

    // limits for counting objects
    float ObjectMinsize;
    float ObjectMaxsize;

    // parameters for use in UDP communication
    char comm_pass[256];
    unsigned int comm_port;
    //int objectamount;

    // chrono time measuring variables
    std::chrono::high_resolution_clock::time_point start_time;
    std::chrono::high_resolution_clock::time_point end_time;
    unsigned int time_dif;

    // udp communication variables
    int sockfd; /* socket file descriptor */
    struct sockaddr_in server_addr, client_addr; // server & client address
    socklen_t clientlen; /* length of client address */
    char CommPassBuffer[64]; /* message received from client */
    char CommSendBuffer[2048]; /* message sent to client */


    // struct for object member variables
    struct Object
    {
        unsigned int index; // object index
        int x; // x coord
        int y; // y coord
        int area; // area value
        unsigned int removcnt; // if this reaches 0, the object is removed
        unsigned int lifecnt; // amount of cycles the object has existed
        float cm; // coordinate margin
        int lhue;
```

```
        int hhue;
        int lsat;
        int hsat;
        int lval;
        int hval;

        Object(unsigned int newindex, int newx, int newy, int newarea, int
rmdef, int hsv[])
        {
            index = newindex;
            x = newx;
            y = newy;
            area = newarea;
            removcnt = rmdef;
            lifecnt = 0;
            cm = (sqrt(area))/2;
            lhue = hsv[0];
            hhue = hsv[1];
            lsat = hsv[2];
            hsat = hsv[3];
            lval = hsv[4];
            hval = hsv[5];
        }

    };

    unsigned int IDcounter; // will have a new ID for each object
    unsigned int rm_default;
    unsigned int MinLife;
    int iObjMove;

    std::vector<Object> vecExistingObjects;
    std::vector<Object> vecFoundObjects;

    /******************** OpenCV-related and other ********************/
    /******************** private access functions ********************/

    // threshold image with user defined parameters
    void ThresholdImage(cv::Mat, cv::Mat&, int [], bool);

    // erode & dilate binary image
    void MorphImage(unsigned int, int, cv::Mat, cv::Mat&);

    // create vectors for moments, areas and mass centers
    int FindObjects(cv::Mat, float, float, std::vector<Object>&);

    // work with object vectors
    unsigned int AddNewObjects(std::vector<Object> found,
std::vector<Object>& exist);
    void ExistentialObjects(std::vector<Object> found, std::vector<Object>&
exist);
    unsigned int CleanupObjects(std::vector<Object>& exist);
    bool FitMargin(int ax, int ay, int bx, int by, float cm);
```

```
    // draw circles around found objects
    void DrawCircles(cv::Mat, cv::Mat&, std::vector<Object>);

    // Information transmission via UDP
    void SetupSocket();/* bind socket */
    void RecvSend(char*, char*); /* receive and send information back (if
correct pass) */
    void WriteSendBuffer(std::vector<Object>, char*); /* write useful
information to buffer */

    /******************** Public access variables *********************/
    /************************ and functions **************************/
    public:

    cv::Mat imgOriginal; /* this Mat is public because it's used in main */

    void setHSV (int *); /* set hue, saturation and light intensity*/

    int* hsv() { return iHSV; } /* return HSV array */

    int val(unsigned int k) { return iHSV[k]; } /* return single value from
hsv array */

    unsigned int height() { return uiCaptureHeight; }

    unsigned int width() { return uiCaptureWidth; }
    bool getGUI() { return bGUI; }

    bool showingImages() { return (iShowOriginal > 0 || iShowThresh > 0); }

    ColourTracking() /* assign default values */
    {
        iCount = ENABLED;
        iMorphLevel = DISABLED;
        iShowOriginal = DISABLED;
        iShowThresh = DISABLED;
        bGUI = ENABLED;
        bThreshBlur = ENABLED;

        int buffer[6] = {LHUE, HHUE, LSAT, HSAT, LVAL, HVAL};
        setHSV(buffer);

        uiDelay = MAX_CYCLE_T - MIN_CYCLE_T;
        iDebugLevel = DEF_DEBUG;
        uiCaptureHeight = CAP_HEIGHT;
        uiCaptureWidth = CAP_WIDTH;

        ResizeImages = false;

        ObjectMinsize = (uiCaptureHeight * uiCaptureWidth) / 100;
        ObjectMaxsize = (uiCaptureHeight * uiCaptureWidth) / 4;

        strncpy(comm_pass, COMM_PASS, sizeof(COMM_PASS));
        comm_port = COMM_PORT;
```

```
        rm_default = 5;
        MinLife = rm_default * 2; // default is always higher than removal
        iObjMove = ENABLED;

        SetupSocket();
    }

    void Process();

    // display original and thresholded images
    void Display();

    // run-time control panel with highgui trackbars
    bool CreateControlWindow();

    // calculate delay for the waitKey function in main.cpp
    unsigned int delay();
    void t_start(); /* set starting timepoint */
    void t_end();   /* set end timepoint and calculate time_dif*/

    // parse command line arguments
    int CmdParameters(int, char**);

    // returns timestamp "[HH:MM:SS]"
    std::string ts();

};


#endif
```

Listing 13. Demonstrative hue bar

```
cv::Mat ColourTracking::DemoHue()
{
      cv::Mat hsvbar = cv::Mat::zeros(100, 180, CV_8UC3);
      cvtColor(hsvbar, hsvbar, CV_BGR2HSV);
      for (int i = 0; i < hsvbar.rows; i++) {
            for (int j = 0; j < hsvbar.cols; j++) {

                  cv::Vec3b newpx;
                  newpx[0] = j;
                  newpx[1] = 255;
                  newpx[2] = 255;
                  hsvbar.at<cv::Vec3b>(i, j) = newpx;

            }
      }
      cv::cvtColor(hsvbar, hsvbar, CV_HSV2BGR);

      return hsvbar;
}
```