Image Analysis with Microcomputer - 30330

Kristian Sloth Lauszus

Kongens Lyngby 2015

Technical University of Denmark
DTU Space
National Space Institute
Elektrovej, building 327-328
2800 Kongens Lyngby, Denmark

Summary (English)

The goal of this report is describe the theory and development of a machine that is able to track and react to objects on a phone screen in real-time. The images are taken using a camera module and is then filtered and analysed using a Raspberry Pi 2 microcomputer. This information will then be used in order to activate one of two solenoids depending on the location of the objects.

The source code for the entire project is available in the Appendix and under the GPLv2 license at the following website: `https://github.com/Lauszus/ImageAnalysisWithMicrocomputer30330/tree/master/ZomBuster`.

Summary (Danish)

Formålet med denne rapport er at beskrive teorien og udviklingen af en maskine der kan tracke og reagere på objekter på en telefon skærm i realtid. Billederne er taget med et kameramodul og filtreret og analyseret vha. en Raspberry Pi 2 microcomputer. Denne information er defter brugt til at aktivere én af to solenoids afhængig af position af objekterne.

Kildekoden for hele projektet er vedlagt i Appendix og tilgængelige under GPLv2 licens på følgende hjemmeside: `https://github.com/Lauszus/ImageAnalysisWithMicrocomputer30330/tree/master/ZomBuster`.

Preface

This report was composed by Kristian Sloth Lauszus (s123808) at DTU Space at the Technical University of Denmark (DTU). This report is made as a 10 ECTS-point project in fulfilment of the compulsory assignment in the course Image Analysis with Microcomputer (30330). The report deals with the theory and development of a simple machine that demonstrates the basic concepts of image analysis and object detection.

In the first part of the report the theory behind the different filters will be introduced and the methods used to calculate the center of mass, contour, Euler number and various others methods will be described. The theory is based on the course book *Robot Vision* by Berthold Klaus Paul Horn[3] additional resources will be highlighted in the text.

The second part uses the theory described in part 1, to implement a system that is able to track zombies in a mobile game and activate two solenoids based on their location.

In the final part the performance will be evaluated, future work is proposed and a conclusion to the project is outlined.

Kristian Sloth Lauszus

Contents

Chapter 1

Introduction

The use of cameras for automatic quality inspection is becoming more and more widespread in the industry. For instance a camera could be used in order to ensure that a product had a certain size, shape and/or colour. This could save a lot of time and money since this no longer has to be carried out by a person.

The purpose of this assignment is to make such a system. However it will be using the mobile game ZomBuster[1] instead of a production environment.

The open source cross-platform library OpenCV which has a lot of different filters and functions built-in. This makes it easy to capture images from a camera and manipulate them. However since one of the goals of this exercises is to implement the different filters ourself, the built-in filters in OpenCV will not be used and is written from scratch.

[1] `https://play.google.com/store/apps/details?id=it.tinygames.zombuster`

Part I

Part 1 - Image theory

Chapter 2

Filters theory

In this chapter some of the basic theory about image filters will be presented and some examples of linear filter kernels will be shown. The different filters discussed in this chapter are all implemented in *filter.cpp* and *filter.h* which is found in Appendix B.

2.1 Binary images

Colour images are typically represented as three channels 8-bit images, i.e. each pixel can have a value between 0-255 for each channel. In the case of RGB images, the first channel indicates the amount of red, the second channel the amount of green and the last channel represents the amount of blue in the pixel.

Another way of representing an image is by using only one channel where the value represent the brightness of the pixel. This is called a greyscale image.

A even simpler image type is a so called binary image where each pixel can only be either 0 or 1. This is useful, as a binary images requires less storage are often faster to manipulate. For instance in the case of object detection, where the object will have one value and the background the other.

In order to convert an image from greyscale to binary format a threshold is chosen and any pixels below this value will be set to one value, while any pixels equal or above the threshold will be set to the other value. In order to chose a threshold value it is often useful to look at the histogram and pick a threshold value that lies in between two hills, as this will separate the object from the background. An example of this will be given in Chapter 3.2.

The code for determining the histogram from an image is shown in Appendix G.

Figure 2.1 shows an example of the effect when an image is converted to greyscale and then to a binary image.

(a) Colour image **(b)** Greyscale image **(c)** Binary image

Figure 2.1: Colour, greyscale, and binary version of an image

2.2 Linear filters

A linear filter consist of a kernel of filter coefficients which are convoluted with an image. The result of such a convolution is the filtered image. The discrete operation is equivalent to a moving window given as:

$$p_{ik} = \sum_{k=i-n}^{i+n} \sum_{l=j-m}^{j+m} c_{kl} q_{kl} \tag{2.1}$$

Where q is a discrete image, c is the kernel coefficients with the size $(2n+1) \times (2m+1)$. The resulting image p will thus be a linear function of its neighbouring $(2n+1) \times (2m+1)$ pixels. One aspect one has to take into account is for instance the top-left pixel has not got any neighbours to the left or on top of itself, thus one has to prevent a overflow situation when implementing the kernel convolution in practice.

Figure 2.2 shows some of the most common linear filter kernels, which are 3×3 filter kernels, and approximate low-pass, high-pass, and Laplacian filtering.

However in order conserve the overall image brightness a filter kernel must satisfy:

$$\sum_{k=-n}^{n} \sum_{l=-m}^{m} c_{kl} = 0 \vee 1 \tag{2.2}$$

As it can be seen the low-pass filter in Figure 2.2a does not fulfil this condition, as the sum is equal to 9. For this reason the filter kernel has to be normalized, that is to divide all filter kernels by the sum i.e. 9 in this case. This is all handled automatically by the *LinearFilter* class, so the user does not have to take this into account manually.

1	1	1
1	1	1
1	1	1

(a) Low-pass kernel

-1	-1	-1
-1	8	-1
-1	-1	-1

(b) High-pass kernel

0	1	0
1	-4	1
0	1	0

(c) Laplacian operator kernel

Figure 2.2: Common linear filter kernels

Thus creating a new linear filter is just as easy as defining a new filter kernel and creating a new instance of the filter using the filter kernel.

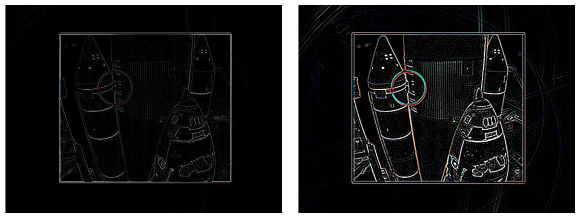
Some common linear filters are already implemented using derived classes, which inheritance the *LinearFilter* class, but calls the constructor with predefined filter kernels. For instances the low-pass, high-pass, and Laplacian kernels shown in Figure 2.2 are all implemented including some more exotic ones like Laplacian-triangular and Laplacian-Gaussian filters.

Figure 2.3 shows the result of several linear filters. As it can be seen the low-pass filter smooth out the image slightly. The high-pass and Laplacian filter on the other hands can be useful for finding the edges of an image. It can be seen that the combination of the applying both a Laplacian and low-pass filter to an image makes the edges stand out even more.

2.2.1 Filter kernel combination

One big advanced of linear filters is that several filter kernels can be combined into one, as shown in Figure 2.4, and thus the filter operation only has to be applied once, but with the combined filter kernel. However the size of the filter kernel will increase as well, but it still is much faster than running all the linear filters one by one.

(a) Original image **(b)** Low-pass filter **(c)** High-pass filter



(d) Laplacian filter **(e)** Laplacian low-pass filter

Figure 2.3: Examples of linear filters

This is all handled by the *LinearFilter* class by overloading the needed operators. For instance in order to obtain the combined Laplacian and low-pass filter kernel shown in Figure 2.4d. All one has to do is the following:

```
static LinearFilter filter = LaplacianFilter() + LowpassFilter();
```

2.3 Fractile filter

Another very useful filter is the fractile filter which assigns the center pixel a value depending on the percentile chosen for the filter. In the special case when the percentile 50 % is chose, the filter is called a median filter.

It works by setting the center pixel to a value that ensures that the center has a brightness that corresponds to the percentile of the original and neighbouring entries. I.e. in the case of a median filter the median of the brightness of neighbouring entries is chosen. For instance if a window size of 3×3 is chosen the brightness's could like:

$$v = [0\,1\,1\,2\,2\,3\,3\,4\,5] \tag{2.3}$$

Thus the brightness is set to $median(v) = 2$ in the case of a median filter.

1	1	1	
1	1	1	
1	1*0	1*1	0
	1	-4	1
	0	1	0

(a) 1. step

1	1	1	
1	1*0	1*1	0
1	1*1	1*-4	1
	0	1	0

(b) 2. step

1*0	1*1	1*0
1*1	1*-4	1*1
1*0	1*1	1*0

(c) 3. step

0	1	1	1	0
1	-2	-1	-2	1
1	-1	0	-1	1
1	-2	-1	-2	1
0	1	1	1	0

(d) Resulting filter kernel

Figure 2.4: Combination of Laplacian and low-pass filter kernels. Only first three steps are shown

Due to the non-linear properties of the fractile filter it has to be run by itself. However it has great advance in certain situations compared to for instance it has better edge conservation, as it does not smear out the image, as the low-pass filter.

It is also particularly good at reducing salt- & pepper noise, as a small amount of pixels affected by noise will completely be removed. This of cause depends on the chosen percentile and window size and will depend on the application.

One approach of implementing the fractile filter is to calculate the histogram of the window chosen and then add up the values in the histogram until the values is equal or more than the desired value. For instance in the case of a median filter and a 3×3 window the position of the is located at $3 \times 3 \times 50\% >= 4.5$, thus by adding up the values in (2.3) the correct value i.e. 2 is obtained.

However as it can take considerable time to calculate the histogram, as the window size increases, it is important to consider how one can optimize the time it takes to calculate the histogram. As the window just shifts one pixel each time, the histogram calculation can be optimized by first removing the leftmost values in the histogram and then adding the new values to the right side of the histogram[2][6].

In the special case where the image is a binary image and the object is solid and is small compared to the image size further optimization can be obtained by skipping a certain pixel value. For instance if the object is white all black

pixels can skipped. This is especially important when one wants to analyse an solid object.

The fractile filter is implemented in *filter.cpp* found in Appendix B.

Figure 2.5 shows how a fractile filter can reduce salt- & pepper noise from an image without distorting the edges of the image.

(a) Original image **(b)** Image after fractile filter

Figure 2.5: Fractile filter example
Original image: https://en.wikipedia.org/wiki/File:Medianfilterp.png

2.4 Morphological filter

Another popular non-linear filter is the morphological filter. The two basic operators are called erosion and dilation operators. Figure 2.6 shows the basic principle of these operators. In this example a square 3×3 structuring element is chosen and applied to the object. As it can be seen the erosion operator simply removes one pixel along the contour of the object while the dilation operator pads the contour with one pixel.

The true advances of the dilation and erosion operators shows when they are combined into what is know as morphological closing and opening. Figure 2.7 shows the result of morphological closing applied to an image. As it can be seen in Figure 2.7b the result of first dilating the image is that the holes in the object are closed. However both the object and the noise has been dilated as a result. This can be undone by eroding the dilated image with the same sized

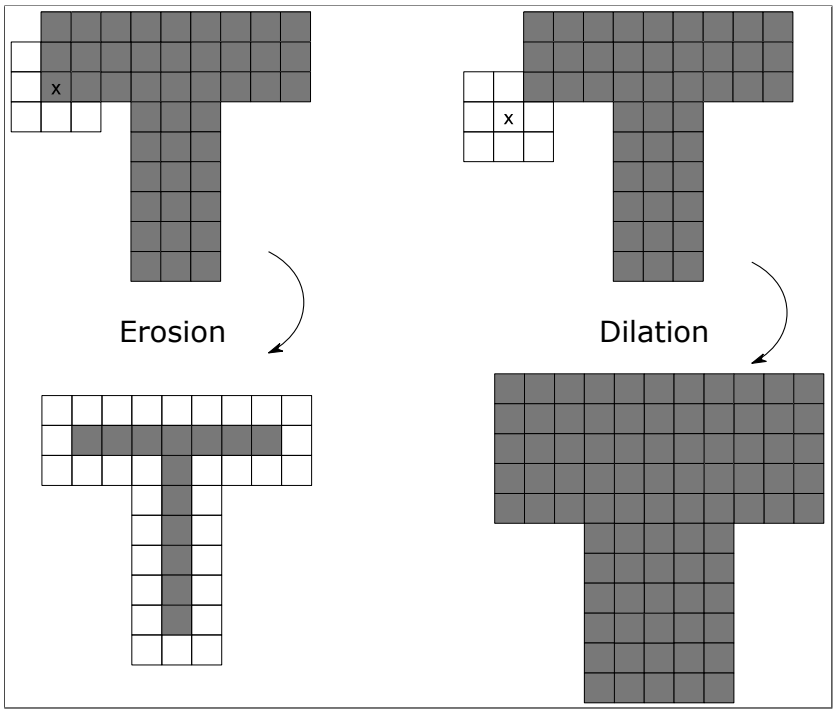


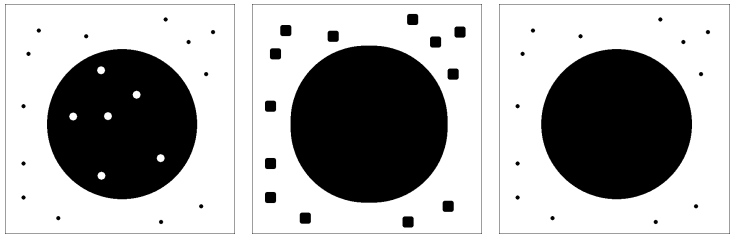
Figure 2.6: Erosion and dilation operators
Credit: Mathias Benn

structuring element as shown in Figure 2.7c. Thus the morphological closing operator will close an object.

The morphological opening operator works quite similar, but instead the image is first eroded and then dilated, as shown in Figure 2.8. The result is that the pepper noise is removed by the erosion and the dilation makes sure that the size of the object and holes are restored.

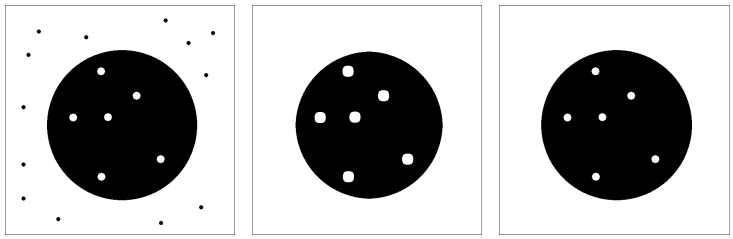
If one applied the morphological opening to the image in Figure 2.7c instead it would be the same as running both morphological closing and opening on the original image. The result of this operation can be seen in Figure 2.9. Thus by applying the morphological closing and the morphological opening operator, both the holes in the object has been closed and the pepper noise around the object has been removed without altering the size of the object[4].

The morphological dilation and erosion are implemented in *filter.cpp* found in Appendix B using a square structuring element with a variable size.



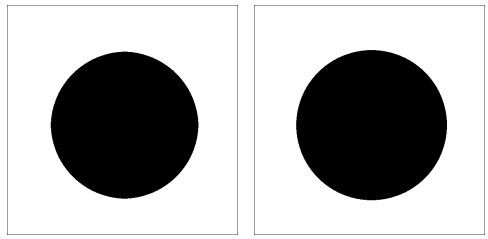
(a) Original image **(b)** Dilated image **(c)** Eroded image

Figure 2.7: Effect of morphological closing



(a) Original image **(b)** Eroded image **(c)** Dilated image

Figure 2.8: Effect of morphological opening



(a) Eroded image **(b)** Dilated image

Figure 2.9: Effect of morphological opening and closing

Chapter 3

Object detection

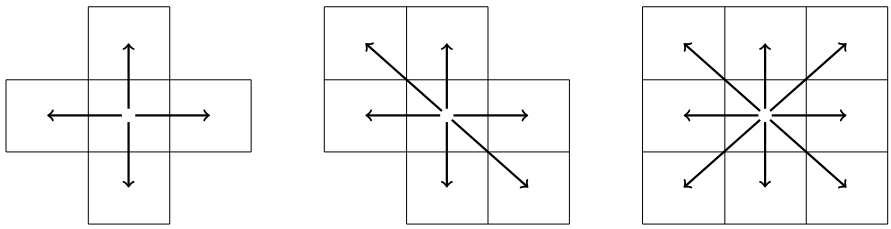
In this chapter some of the different methods that can later be used for object segmentation and detection will be presented. This is useful in order to differentiate different objects from each other and then analyse the objects separately.

3.1 Segmentation

In order to split up an image in different objects it is important to define when two pixels are actually connected to one another. Figure 3.1 shows the three common form of connectedness. As it can be seen 4-connectedness only looks at the pixels above, below, left and right of the center pixel. 6-connectedness includes the top-left and bottom-right corner while the 8-connectedness includes all pixels around the center pixel.

Figure 3.2 shows an example of segmentation using 4- and 8-connectedness on a binary image. As it can be seen two segments are identified when using 4-connectedness while only one segment is found when using 8-connectedness.

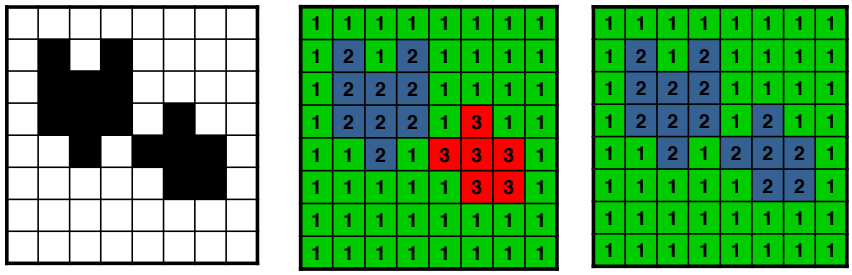
The source code for finding segments in an image is found in Appendix C. This approach uses 8-connectedness and will take an image as input and return an



(a) 4-connectedness **(b)** 6-connectedness **(c)** 8-connectedness

Figure 3.1: Connectedness

array of images where each image represent one segment. Furthermore it is also possible to adjust the neighbour size i.e. how many pixels around the center pixel it should look at. This turned out to be useful in practice if the image is effected by noise.



(a) Binary image **(b)** 4-connectedness **(c)** 8-connectedness

Figure 3.2: Segmentation using 4- and 8-connectedness
Credit: Troelz Denver

3.1.1 Contour search

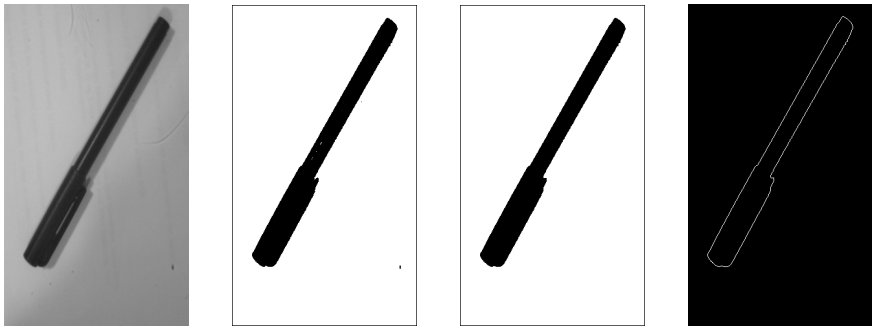
In order to find the contour of an image one could use a Laplacian filter as described in chapter 2.2. Another approach is to use a iterative contour search where the contour of an image is found by first finding the border of the object. The next search direction will then be just above the current pixel according to the previous search direction. If their is no pixel set at that location, the next pixel is searched. If 8-connectedness is used it will use the corner pixel while if 4- or 6-connectedness is used it will look at the pixel in front of the current pixel. This is repeated around the entire pixel according to the connectedness

used until the next contour pixel is found and the procedure is repeated. Once the original pixel is found again, the final contour is found.

It is important to note that this approach is actually faster if their is only one object compared to for instance a Laplacian filter, as it will not waste time going through all pixels in the entire image, but only the pixels in the contour. However this approach only works for binary images.

Figure 3.3 shows four steps used for finding the contour of an image. First the original image is thresholded based on the histogram, then a morphological filter is used in order to fill out the gabs in the image and remove noise sources, like the dot in the bottom right of 3.3b. Finally the contour of the image is found by using the iterative contour search.

The source code for finding the contour of an image can be found in *contour.cpp* in Appendix D.



(a) Original image **(b)** Thresholded image **(c)** Morphological filtering **(d)** Found contour of image

Figure 3.3: Steps in order to find the contour of an image

3.2 Moments

In order to differentiate objects from one and another it can be useful to calculate the moment of an object, as each object will have different values, that allows one to distinguish between them.

In order to calculate the $j+k$ order of moment on can use the following equation:

$$M_{jk} = \sum_x \sum_y x^j y^k f(x,y)\,, \quad j+k = 0,1,2,3\ldots \tag{3.1}$$

Where $f(x,y)$ is the image description of the binary image i.e. it will be either 0 or 1 depending on the current values of x and y.

It can be noted that the area will be the 0-order moment and is simply given by:

$$Area = M_{00} \tag{3.2}$$

Furthermore one can easily calculate the center of mass of an object once the 0-order and 1-order moments are calculated:

$$\bar{x} = \frac{M_{10}}{M_{00}} \tag{3.3}$$

$$\bar{y} = \frac{M_{01}}{M_{00}} \tag{3.4}$$

However these moments will depend on the current position of the object, instead one can calculate what is called central moments which are translational invariant, thus they will not change when the object are moved in the x,y-plane:

$$\mu_{jk} = \sum_x \sum_y (x-\bar{x})^j (y-\bar{y})^k f(x,y)\,, \quad j+k = 0,1,2,3\ldots \tag{3.5}$$

It can be shown that the first six central moments can be simplified to:

$$\mu_{00} = M_{00} \tag{3.6}$$

$$\mu_{10} = \mu_{01} = 0 \tag{3.7}$$

$$\mu_{11} = M_{11} - \bar{y}M_{10} = M_{11} - \bar{x}M_{01} \tag{3.8}$$

$$\mu_{20} = M_{20} - \bar{x}M_{10} \tag{3.9}$$

$$\mu_{02} = M_{02} - \bar{y}M_{01} \tag{3.10}$$

These can also be used to calculate the angle of an object according to the following equation:

$$\theta = 0.5\,\mathrm{atan2}(2\mu_{11}, \mu_{20} - \mu_{02}) \tag{3.11}$$

Where atan2 is like the ordinary atan function, but it takes the sign of the two arguments into account, so the quadrant of the computed angle is correctly computed[1].

Figure 3.4 shows an example of a pen where the center of mass and angle of the object has been calculated and drawn on the image. The histogram to the right shows the current threshold value used to convert the greyscale image into a binary image.

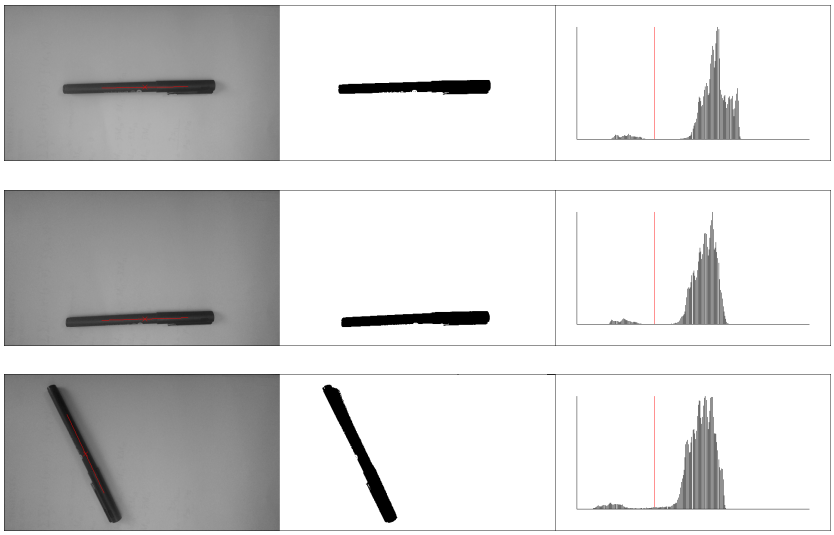


Figure 3.4: Center of mass and the angle of an object

Scale invariant moments are as the name suggest also independent on scaling of an object i.e. movement in the z-plane and are given by:

$$\eta_{jk} = \frac{\mu_{jk}}{{\mu_{00}}^{\gamma}}, \quad j+k = 2, 3\ldots \tag{3.12}$$

$$\gamma = \frac{j+k}{2} + 1 \tag{3.13}$$

These however are not independent of rotation. The so called Hu set of invariant moments on the other hand are both independent on translation, scale, and rotation.

The first two Hu set of invariant moments are given as:

$$\phi_1 = \eta_{20} + \eta_{02} \tag{3.14}$$

$$\phi_2 = (\eta_{20} - \eta_{02})^2 + 4{\eta_{11}}^2 \tag{3.15}$$

The Hu set of invariant moments are especially helpful if one want to track a certain object in real time that is able to move freely in all three dimensions.

The different moments described in this chapter can be calculated using the code found in Appendix F.

3.3 Euler number

Another useful mathematical number is the Euler number, which is given by:

$$E = C - H \tag{3.16}$$

Where C is the number of connected objects and H is the number of holes. Thus if there is only one object with no holes, the Euler number will be equal to 1.

For instance the Euler number of Figure 2.7a would be:

$$E = 14 - 6 = 8 \tag{3.17}$$

However the Euler number of the same image after morphological opening and closing, as shown in Figure 2.9b, will be:

$$E = 1 - 0 = 1 \tag{3.18}$$

Thus if the goal was to locate one solid object the Euler number would be a quick way to check if filtering is needed. However the Euler number would not be able to distinguish between one solid object and two objects with one hole.

Similarly to segmentation the Euler number can be calculated in different ways depending on connectedness used.

Thus for 4-connectedness the Euler number is given as:

$$E = \frac{n(Q_1) - n(Q_3) + 2\,n(Q_D)}{4} \tag{3.19}$$

While for 8-connectedness it is given by:

$$E = \frac{n(Q_1) - n(Q_3) - 2\,n(Q_D)}{4} \tag{3.20}$$

Where $n(Q_x)$ is the number of bit quad Q_x, as shown in Figure 3.5. Thus by counting the number of occurrences of a particular bit quad the Euler number in a binary image can easily calculated according to the two equations.

The source code for calculating the Euler number of a binary image can be found in Appendix E.

$$Q_0 = \begin{array}{|c|c|}\hline 0 & 0 \\ \hline 0 & 0 \\ \hline \end{array} \quad Q_4 = \begin{array}{|c|c|}\hline 1 & 1 \\ \hline 1 & 1 \\ \hline \end{array} \quad Q_D = \begin{array}{|c|c|}\hline 1 & 0 \\ \hline 0 & 1 \\ \hline \end{array}, \begin{array}{|c|c|}\hline 0 & 1 \\ \hline 1 & 0 \\ \hline \end{array}$$

$$Q_1 = \begin{array}{|c|c|}\hline 1 & 0 \\ \hline 0 & 0 \\ \hline \end{array}, \begin{array}{|c|c|}\hline 0 & 1 \\ \hline 0 & 0 \\ \hline \end{array}, \begin{array}{|c|c|}\hline 0 & 0 \\ \hline 1 & 0 \\ \hline \end{array}, \begin{array}{|c|c|}\hline 0 & 0 \\ \hline 0 & 1 \\ \hline \end{array}$$

$$Q_2 = \begin{array}{|c|c|}\hline 1 & 0 \\ \hline 1 & 0 \\ \hline \end{array}, \begin{array}{|c|c|}\hline 1 & 1 \\ \hline 0 & 0 \\ \hline \end{array}, \begin{array}{|c|c|}\hline 0 & 1 \\ \hline 0 & 1 \\ \hline \end{array}, \begin{array}{|c|c|}\hline 0 & 0 \\ \hline 1 & 1 \\ \hline \end{array}$$

$$Q_3 = \begin{array}{|c|c|}\hline 1 & 1 \\ \hline 1 & 0 \\ \hline \end{array}, \begin{array}{|c|c|}\hline 1 & 1 \\ \hline 0 & 1 \\ \hline \end{array}, \begin{array}{|c|c|}\hline 0 & 1 \\ \hline 1 & 1 \\ \hline \end{array}, \begin{array}{|c|c|}\hline 1 & 0 \\ \hline 1 & 1 \\ \hline \end{array}$$

Figure 3.5: Bit quad patterns

3.4 HSV color space

If one wants to detect a object with a certain colour it might be difficult to achieve this using the RGB (Red, Green, Blue) color space, as certain nuances of a certain color can have non-intuitive values. Instead it is more useful to use the HSV (Hue, Saturation, Value) color space where the Hue value represent a color including all possible tints, tones and shades of that particular color. Saturation describes the intensity of the color i.e. a saturation of 0 is white and the maximum saturation is the fully saturated colour. The value specifies the amount of lightness, where 0 is black and increasing lightness moving away from black. Figure 3.6 shows a graphical representation of the HSV color space[5][7].

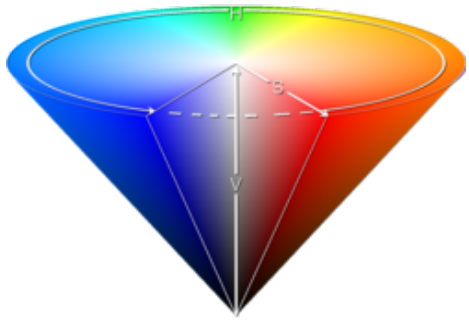
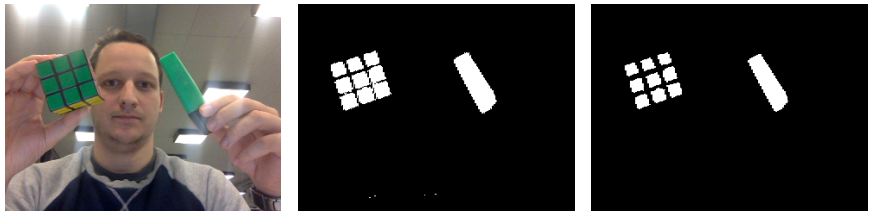


Figure 3.6: HSV color cone
Original image: http://colorizer.org/img/hsv.png

3.5 Object detection

Converting an image into HSV color space allows an object with a certain color to be easily identified, as it is easy to threshold such an image by a human, as one could just look for a particular range of hue values and set a lower limit of the saturation and value. Figure 3.7 shows an example of where this approach has been applied to an image. Furthermore a fractile filter has been applied in order to remove some salt & pepper noise from the image.

Let say we want to track the Rubik's Cube. In order to do so a morphological closing and opening is first applied to the image in order to close the holes between the squares in the Rubik's Cube. After that the two objects are separated into to separate images using the segmentation method described in 3.1.



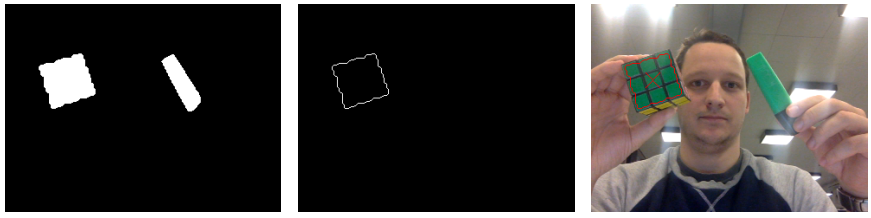
(a) Original image **(b)** HSV thresholding **(c)** Fractile filter

Figure 3.7: HSV thresholding

The first Hu set of invariant moments is then calculated for each object and compared to a certain predefined threshold furthermore the Euler number should be equal to 1. If both of these conditions are true the center of mass and the contour of the image are superimposed on the original image.The contour is found using iterative contour search method described earlier in this chapter.

As it can bee seen this method can successfully identify the Rubik's Cube as seen in Figure 3.8.

Note since the first Hu set of invariant moment is used it can successfully track the object even though it is moved in all three dimensions. Also since the different objects are segmented this method also allows one to track several objects at once.



(a) Morphological filter **(b)** Moment thresholding & contour search **(c)** Found object

Figure 3.8: Object detection

Part II

Part 2 - Design & implementation

Chapter 4

Design and implementation

In this part the different filters and method described in the theory will be used in order to track Zombies in the mobile game ZomBuster[1] using a Raspberry Pi 2 and a camera module. This information will then be used to kill the zombies depending on their location on the screen by activating either a left or right solenoid. The entire source code is attached in the appendices or at the following website: `https://github.com/Lauszus/ImageAnalysisWithMicrocomputer30330/tree/master/ZomBuster`.

The challenge might seem a little silly at first, and it properly is, but the same principles could easily be applied to a lot of other scenarios. For example it could be used for automatic quality inspection in a production environment e.g. be used to discard malformed cookies, plastic parts etc. An air-gun could then be used to shoot off that particular item.

4.1 ZomBuster

A screenshot of the mobile game ZomBuster is shown in Figure 4.1. The goal of the game is to shoot the Zombie in either the left or right lane. Each time a

[1] `https://play.google.com/store/apps/details?id=it.tinygames.zombuster`

zombie is killed the score increases by one. As the score increases the speed of of the game increases as well making it more and more difficult. One can lose the game by either shooting a human or if a zombie reaches the bottom of the screen.

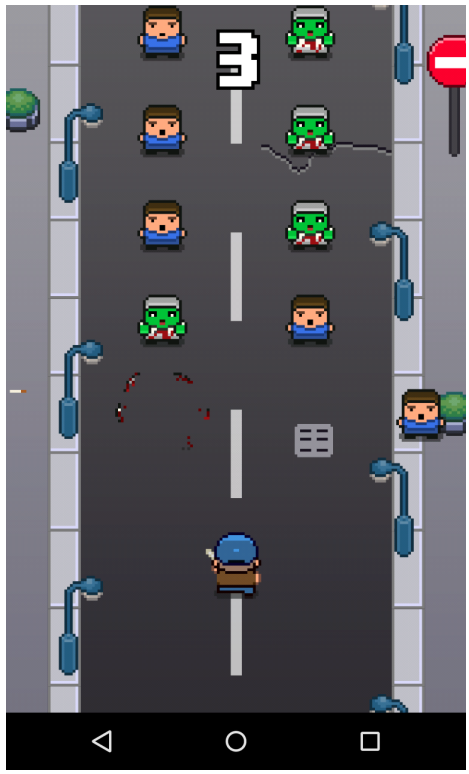


Figure 4.1: ZomBuster gameplay

In order to kill the zombies two solenoids are positioned at the left and right side of the screen in an appropriate hight so the end of the rod touches the screen when the solenoids is activated. A tip of a tablet stylus is glued onto the end of each solenoid's rod in order not to damage the screen.

4.2 Hardware considerations

I decided to use a Raspberry Pi 2 which features a Broadcom BCM2836 ARM Cortex-A7 quad core CPU running at 900 MHz and 1 Gb of RAM. Furthermore

an inexpensive 5 mega pixel omnivision 5647 camera module[2] is used which can record up to 90 FPS at 640×480 pixels. However the Raspberry Pi will not be able to run that fast when doing the image processing even though the image size were decreased to only 160×120 pixels.

Also in order to decrease the number of variables the exposure, ISO, white balance etc. were set to a fixed value, so the driver does not adjust it dynamically. Furthermore the phone screen's brightness setting was set to maximum and the LED on the camera module was turned in order to reduce reflections.

Also in order to control the GPIO pins on the Raspberry Pi 2 the WiringPI[3] library is used, this allows one to control and setup the GPIO pins directly in the code. Also I used the UV4L driver[4] in order to access the camera module using the OpenCV API.

By using a camera that was able to shoot at 240 FPS I measured the minimum period were the solenoids would still work reliably. This was measured to be around 60 ms or around 17 FPS.

Since I decided to use an ARM platform there is no pre-compiled version of OpenCV available, so I had to compile the source code myself. A script can be found at the the following link: `https://gist.github.com/Lauszus/c8ce73f3177d6455c27c#file-install_opencv-sh` which makes it easy to compile and install OpenCV on the Raspberry Pi 2 easily.

4.2.1 Schematic

The circuit used to drive the two solenoids is shown in Figure 4.2. It takes two inputs from the Raspberry Pi and then uses a BC547 NPN transistor in order to level-shift the voltage from 3.3V to 12V. This signal is then connected to the gate on a IRF3205 N-Channel Mosfet which drives the solenoid. Note the end result is that the output is inverted, thus a high signal needs to be applied to the base of the BC547 in order for the IRF3205 to turn off. For that reason the base of the BC547 is pulled high to 3.3V using a $100k\Omega$ resistor, so the IRF3205 will is off by default.

Furthermore a button is connected to ground which allows the user to start and

[2] `http://uk.rs-online.com/web/p/video-modules/7757731`

[3] `http://wiringpi.com`

[4] `http://www.linux-projects.org/modules/sections/index.php?op=viewarticle&artid=14`

stop the program using a script[5] without the need to connect a keyboard to the Raspberry Pi. Since the button has no pull-up resistor connect the internal pull-up resistor must be activated on the input pin on the Raspberry Pi in order to prevent the input-pin from floating.

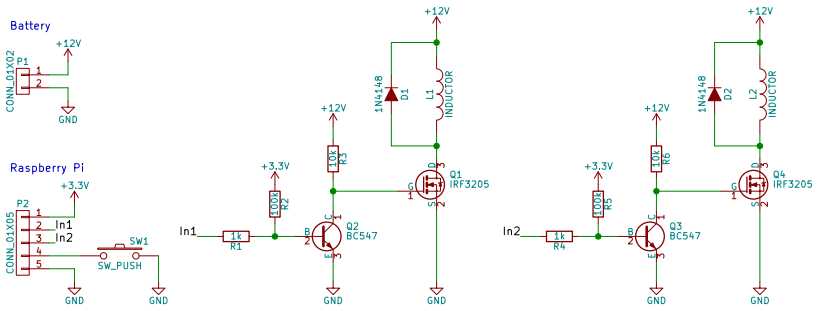


Figure 4.2: Schematic

4.3 3D model

In order to hold the different components I decided to make a 3D model and use a 3D printer in order to print the different components. Figure 4.3 shows a rendering of the finished 3D model. The 3D printed parts consist of the phone holder, back-panel, solenoid holder and camera arm. The solenoid holder has adjustable slots, so the high of the solenoids can be adjusted. The 3D model is available at the following website: `https://github.com/Lauszus/ImageAnalysisWithMicrocomputer30330/tree/master/ZomBuster/3DModel`.

Figure 4.4 shows a picture of the assembled hardware. To start and stop the program one can simply press the button on the stripboard.

[5] `https://gist.github.com/Lauszus/c8ce73f3177d6455c27c#file-zombuster-sh`

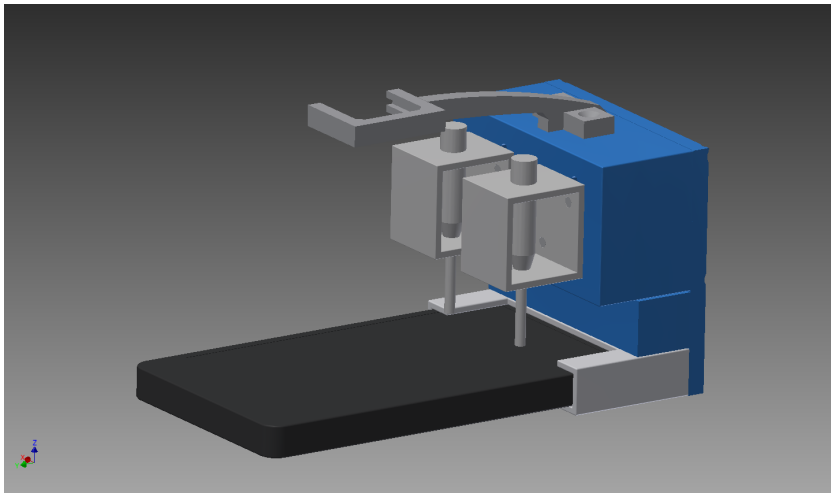


Figure 4.3: 3D model

4.4 Implementation

The same approach as in chapter 3.5 was used in order identify the zombies. Furthermore since there is also sometimes plants in the border of the game as it can be seen in Figure 4.1 the border is ignored in the code. A constrain on the area was added as well because of the animation when the zombies dies, as they will expand and blow up when they are shot.

All moments of these object are stored in an array and sorted in ascending order according to the x-coordinate of the center of mass $\bar{x}$. This is used in order to

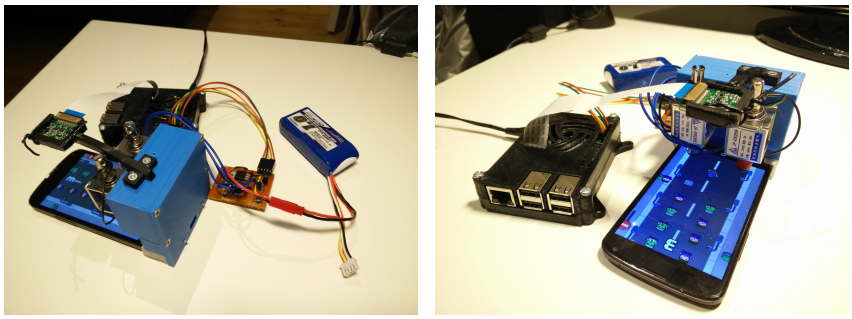


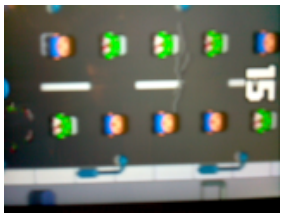
Figure 4.4: Assembled hardware

determine the order of the zombies from the bottom up. Note the camera is tilted 90 degrees in order to increase the field of view, as the camera image is wider than it is tall. The y-coordinate of the center of mass $\bar{y}$ is then used in order to determine if the zombie is either at the left or right side.

If a zombie is detected in one of the sides the code will write a number in a ring buffer. If a zombie is detected at the right side 1 will be written. If the zombies on the other hand is detected at the left side -1 will be written to the ring buffer.

A statemachine is then called which takes care of driving the solenoids. Thus it will first read the ring buffer and then activate the appropriate solenoid. The next time the statemachine is called it will check to see if it more than 30 ms since the solenoid is activated. If that is the case it will deactivate the solenoid. Next time the statemachine is called it will make sure that is more than 30 ms since the solenoid was deactivated, as it has to wait for it to go all the way up again. This will repeat until the ring buffer is empty.

The main code will wait looking for more zombies until the solenoids have finished. A small delay is added after the solenoids are done because it takes some time for the zombies to disappear because of the animation when they are shot.

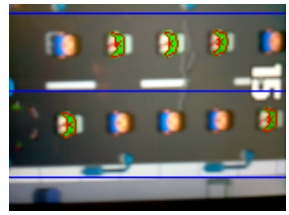


(a) Original image

(b) HSV thresholding

(c) Fractile filter

(d) Morphological filter



(e) Found objects including borders

Figure 4.5: Object detection

Figure 4.5 shows an image of what the Raspberry sees doing a game. It can

be seen that it can successfully identify the five zombies on the screen while still ignoring parts of the zombie that is exploding. Figure 4.5e shows how the screen is split up into sections in software, thus it will ignore the borders of the screen. Furthermore the center line is marked and roughly follows the stripes on the road.

The main code and the ring buffer code can be found in Appendix A and H respectively.

Part III

Part 3 - Evaluation & conclusion

Chapter 5

Results & discussion

The maximum score the machine was able to achieve was 91. For comparison I was able to get a score of 82 with a little practice, thus the machine is able to outperform me, but not with as large a margin as I had hoped. One of the possible reasons is that the Raspbery Pi 2 is only able to run the object recognition code at about 30 FPS even thought the image from the camera were reduced to only 160×120. A more powerful processor might be able to achieve a higher score. Furthermore the code actually waits until the solenoids are done until it tries to kill more zombies. This could properly be improved by some kind of trajectory estimation algorithm, so the future position of the zombies can be estimated. This could also help with the fact that it takes some time after a zombie is detected until it is actually killed, due to the time it takes for the solenoids to hit the screen and because of the kill animation. However as each zombies is killed the screen moves down in a rapid movement which might complicate things.

Two videos of the assembled machine playing the game can be seen at the following website: `https://www.youtube.com/playlist?list=PLRBI0ZWd8RfBdsdsMZBNSGHwYNOCTvplc`. By looking at the slow motion video the delay between when the solenoid hits the screen until the zombies actually disappears is clearly visible.

I am content with the final result, as the machine is able to demonstrate the

basic of object tracking and is able to use this information to respond in real time. Furthermore it was able to outperform me which was one of my personal goals. The project could properly be improved by using a faster processor or implement the code directly on a FPGA or a processor that is not running a full operating system. This should be possible with not that much extra work, as all the different filters are written from scratch and does not rely on the OpenCV filter functions. However one has to come up with a replacement for the *Mat* class which is used throughout the code, as this makes it easy to read and write to an image.

One other aspect is that the camera vibrates a lot when the solenoids are actuated, as the arm holding the camera is attached to the same structure as the one holding the solenoids. This could be solved by printing a new part which grabs on to the phone from the other side.

Chapter 6

Conclusion

This report has examined the theory and implementation of a machine that is able to track and react to objects in real-time.

First the theory and methods needed were described and demonstrated using a static scenario.

The code were then expanded to track and react to zombies in a mobile game. This was then compiled on a Raspberry Pi 2 which activated one of two solenoids depending on their location.

The final machine worked just fine and was able to outperform a human. The project could be properly be improved by using a more powerful processor or even a FPGA.

Bibliography

[1] cplusplus. atan2 - C++ Reference. http://www.cplusplus.com/reference/cmath/atan2, 2015. Visited: 15. December 2015.

[2] R. P. W. Duin, H. Haringa, and R. Zeelen. Fast percentile filtering. In . Pattern Recognition Letters, October 1985.

[3] Berthold Klaus Paul Horn. *Robot Vision*. MIT Press, 1st edition edition, 1986. ISBN-13: 978-0-262-008159-7.

[4] Scikit image development team. Morphological Filtering. http://scikit-image.org/docs/dev/auto_examples/applications/plot_morphology.html. Visited: 15. December 2015.

[5] Sebastian Loncar. http://colorizer.org. Visited: 18. December 2015.

[6] Simon Perreault and Patrick Hébert. Median Filtering in Constant Time. In . IEEE, September 2007. ISSN: 1057-7149.

[7] John W. Shipman. The hue-saturation-value (HSV) color model. http://infohost.nmt.edu/tcc/help/pubs/colortheory/web/hsv.html, 2012. Visited: 18. December 2015.

Part IV

Part 4 - Appendix

Appendix A

Main code

```
/* Copyright (C) 2015 Kristian Sloth Lauszus. All rights reserved.

 This software may be distributed and modified under the terms of the GNU
 General Public License version 2 (GPL2) as published by the Free Software
 Foundation and appearing in the file GPL2.TXT included in the packaging of
 this file. Please note that GPL2 Section 2[b] requires that all works based
 on this software must also be made publicly available under the terms of
 the GPL2 ("Copyleft").

 Contact information
 -------------------

 Kristian Sloth Lauszus
 Web      :  http://www.lauszus.com
 e-mail   :  lauszus@gmail.com
*/

#include <opencv2/highgui.hpp>
#include <opencv2/imgproc.hpp>

#include "contours.h"
#include "euler.h"
#include "filter.h"
#include "histogram.h"
#include "moments.h"
#include "segmentation.h"

#define WRITE_IMAGES    0
#define PRINT_TIMING    0
#define PRINT_FPS       0

#define FPS_MS (1.0/50.0*1000.0) // 50 FPS
```

```
33
34 using namespace cv;
35
36 static bool valueChanged;
37 static void valueChangedCallBack(int pos) {
38     valueChanged  = true;
39 }
40
41 #if __arm__
42 #include <wiringPi.h> // GPIO access library for the Raspberry Pi
43 #include "ringbuffer.h" // Ring buffer class
44
45 RingBuffer zombieBuffer;
46
47 static const uint8_t leftSolenoidPin = 9, rightSolenoidPin = 7, buttonPin =
       8; // These first two pins control the two solenoids the last is a
       button input
48
49 enum Solenoid_e {
50     SOLENOID_READ,
51     SOLENOID_KILL,
52     SOLENOID_LIFT,
53 };
54
55 static bool solenoidDone;
56
57 static void runSolenoidStateMachine(void) {
58     static Solenoid_e state = SOLENOID_READ;
59     static int val;
60     static double timer;
61
62     switch (state) {
63         case SOLENOID_READ:
64             if (((double)getTickCount() - timer) / getTickFrequency() *
                    1000.0 > 30) { // Wait 30 ms for solenoid to go up again
65                 solenoidDone = true;
66                 if (zombieBuffer.available()) {
67                     val = zombieBuffer.read();
68                     state = SOLENOID_KILL;
69                     solenoidDone = false;
70                 }
71             }
72             break;
73         case SOLENOID_KILL:
74             if (val > 0) // Right side
75                 digitalWrite(rightSolenoidPin, LOW); // Kill zombie on the
                        right side
76             else // Left side
77                 digitalWrite(leftSolenoidPin, LOW); // Kill zombie on the
                        left side
78             timer = (double)getTickCount();
79             state = SOLENOID_LIFT;
80             break;
81         case SOLENOID_LIFT:
82             if (((double)getTickCount() - timer) / getTickFrequency() *
                    1000.0 > 30) { // Wait 30 ms for solenoid to go all the way
                    down
83                 if (val > 0) // Right side
84                     digitalWrite(rightSolenoidPin, HIGH);
85                 else // Left side
86                     digitalWrite(leftSolenoidPin, HIGH);
87                 timer = (double)getTickCount();
88                 state = SOLENOID_READ;
89             }
90             break;
```

```
91      }
92  }
93  #endif
94
95  int main(int argc, char *argv[]) {
96      static const bool DEBUG = argc == 2 ? argv[1][0] == '1' : false; // Check
            if DEBUG flag is set
97
98  #if 0 // Green rubiks
99      int iLowH = 60;
100     int iHighH = 100;
101
102     int iLowS = 100;
103     int iHighS = 255;
104
105     int iLowV = 10;
106     int iHighV = 255;
107
108     int closingSize = 20;
109     int openingSize = 3;
110
111     int windowSize = 3;
112     int percentile = 20;
113
114     int neighborSize = 25;
115
116     int objectMin = 1600;
117     int objectMax = 1750;
118
119     int cropPadding = 30;
120
121     int areaMin = 0; // Can be any size
122     int areaMax = (uint16_t)~0;
123 #else // ZomBuset
124     int iLowH = 40;
125     int iHighH = 80;
126
127     int iLowS = 145;
128     int iHighS = 255;
129
130     int iLowV = 55;
131     int iHighV = 255;
132
133     int closingSize = 3;
134     int openingSize = 1;
135
136     int windowSize = 3;
137     int percentile = 50;
138
139     int neighborSize = 5;
140
141     int objectMin = 2100;
142     int objectMax = 3200;
143
144     int cropPadding = 30;
145
146     int areaMin = 50;
147     int areaMax = 100;
148 #endif
149
150     if (DEBUG) {
151         const char *controlWindow = "Control";
152         cvNamedWindow(controlWindow, CV_WINDOW_AUTOSIZE); // Create a window
                called "Control"
153
```

```
154         // Create trackbars in "Control" window
155         cvCreateTrackbar("LowH", controlWindow, &iLowH, 179,
                valueChangedCallBack); // Hue (0 - 179)
156         cvCreateTrackbar("HighH", controlWindow, &iHighH, 179,
                valueChangedCallBack);
157
158         cvCreateTrackbar("LowS", controlWindow, &iLowS, 255,
                valueChangedCallBack); // Saturation (0 - 255)
159         cvCreateTrackbar("HighS", controlWindow, &iHighS, 255,
                valueChangedCallBack);
160
161         cvCreateTrackbar("LowV", controlWindow, &iLowV, 255,
                valueChangedCallBack); // Value (0 - 255)
162         cvCreateTrackbar("HighV", controlWindow, &iHighV, 255,
                valueChangedCallBack);
163
164         cvCreateTrackbar("Closing size", controlWindow, &closingSize, 50,
                valueChangedCallBack);
165         cvCreateTrackbar("Opening size", controlWindow, &openingSize, 50,
                valueChangedCallBack);
166
167         cvCreateTrackbar("Window size", controlWindow, &windowSize, 10,
                valueChangedCallBack);
168         cvCreateTrackbar("Percentile", controlWindow, &percentile, 100,
                valueChangedCallBack);
169
170         cvCreateTrackbar("Neighbor size", controlWindow, &neighborSize, 50,
                valueChangedCallBack);
171
172         cvCreateTrackbar("Object min", controlWindow, &objectMin, 4000,
                valueChangedCallBack);
173         cvCreateTrackbar("Object max", controlWindow, &objectMax, 4000,
                valueChangedCallBack);
174
175         cvCreateTrackbar("Crop padding", controlWindow, &cropPadding, 100,
                valueChangedCallBack);
176
177         cvCreateTrackbar("Area min", controlWindow, &areaMin, 200,
                valueChangedCallBack);
178         cvCreateTrackbar("Area max", controlWindow, &areaMax, 200,
                valueChangedCallBack);
179     }
180
181     VideoCapture capture(0); // Capture video from webcam
182     if (!capture.isOpened()) {
183         printf("Could not open Webcam\n");
184         return 1;
185     }
186     capture.set(CV_CAP_PROP_FRAME_WIDTH, 320);
187     capture.set(CV_CAP_PROP_FRAME_HEIGHT, 240);
188
189 #if __arm__
190     if (wiringPiSetup() == -1) { // Setup WiringPi
191         printf("wiringPiSetup failed!\n");
192         return 1;
193     }
194     pinMode(leftSolenoidPin, OUTPUT); // Set both pins to output
195     pinMode(rightSolenoidPin, OUTPUT);
196     digitalWrite(leftSolenoidPin, HIGH); // Input is inverted, so HIGH
            disables the solenoid
197     digitalWrite(rightSolenoidPin, HIGH);
198     pinMode(buttonPin, INPUT);
199     pullUpDnControl(buttonPin, PUD_UP); // Enable pull-up resistor
200     solenoidDone = true;
201
```

```
202     printf("Ready to kill some zombies!\n");
203     while (digitalRead(buttonPin)) { // Quit if button is pressed
204 #else
205     while (cvWaitKey(1) != 27) { // End if ESC is pressed
206 #endif
207         if (DEBUG && valueChanged) {
208             valueChanged = false;
209             printf("HSV: %u %u\t%u %u\t%u %u\t\tSize: %d %d\tFractile filter:
                      %d %d\tNeighbor size: %d\tObject: %d %d\tPadding: %d\tArea:
                      %d %d\n", iLowH, iHighH, iLowS, iHighS, iLowV, iHighV,
                    closingSize, openingSize, windowSize, percentile,
                    neighborSize, objectMin, objectMax, cropPadding, areaMin,
                    areaMax);
210         }
211
212         double startTimer = (double)getTickCount();
213 #if PRINT_TIMING
214         double timer = startTimer;
215 #endif
216         Mat image;
217         if (!capture.read(image)) {
218             printf("Could not read Webcam\n");
219             return 1;
220         }
221         //flip(image, image, 1); // Flip image so it acts like a mirror
222 #if __arm__
223         resize(image, image, image.size() / 2); // Make image even smaller on
                ARM platforms
224 #endif
225 #if PRINT_TIMING
226         printf("Capture = %f ms\t", ((double)getTickCount() - timer) /
                getTickFrequency() * 1000.0);
227         timer = (double)getTickCount();
228 #endif
229
230     #if 0
231         printf("image size: %lu, width: %d, height: %d, %lu, color format: %d
                \n",
232                 image.total() * image.channels(), image.size().width, image.
                        size().height, image.total(), image.channels());
233     #endif
234         //imshow("Image", image);
235 #if WRITE_IMAGES
236         imwrite("img/image.png", image);
237 #endif
238
239         Mat image_hsv;
240         cvtColor(image, image_hsv, COLOR_BGR2HSV); // Convert image to HSV
241 #if PRINT_TIMING
242         printf("HSV = %f ms\t", ((double)getTickCount() - timer) /
                getTickFrequency() * 1000.0);
243         timer = (double)getTickCount();
244 #endif
245
246         Mat imgThresholded(image.size(), CV_8UC1);
247         Scalar low = Scalar(iLowH, iLowS, iLowV);
248         Scalar high = Scalar(iHighH, iHighS, iHighV);
249
250         // Threshold the image
251         size_t index = 0;
252         for (size_t i = 0; i < image_hsv.total(); i++) {
253             bool inRange = true;
254             for (uint8_t j = 0; j < image_hsv.channels(); j++) {
255                 uchar value = image_hsv.data[index + j];
256                 if (j != 0 || low.val[j] < high.val[j]) {
```

```
257                     if (value < low.val[j] || value > high.val[j])
258                         inRange = false;
259                 } else if (value < low.val[j] && value > high.val[j]) //
                        Needed for red color where H value will wrap around
                        [170;10]
260                     inRange = false;
261             }
262             imgThresholded.data[i] = inRange ? 255 : 0; // Draw thresholded
                    object white
263             index += image_hsv.channels();
264         }
265
266 #if PRINT_TIMING
267         printf("Threshold = %f ms\t", ((double)getTickCount() - timer) /
                getTickFrequency() * 1000.0);
268         timer = (double)getTickCount();
269 #endif
270         //imshow("Thresholded image", imgThresholded);
271 #if WRITE_IMAGES
272         imwrite("img/imgThresholded.png", imgThresholded);
273 #endif
274
275         // Apply fractile filter to remove salt- and pepper noise
276         Mat fractileFilterImg = fractileFilter(&imgThresholded, windowSize,
                percentile, true);
277 #if PRINT_TIMING
278         printf("Fractile filter = %f ms\t", ((double)getTickCount() - timer)
                / getTickFrequency() * 1000.0);
279         timer = (double)getTickCount();
280 #endif
281         //imshow("Fractile filter", fractileFilterImg);
282 #if WRITE_IMAGES
283         imwrite("img/fractileFilterImg.png", fractileFilterImg);
284 #endif
285
286 #if 1
287         // Crop image, so we are only looking at the actual data
288         index = 0;
289         int minX, maxX, minY, maxY;
290         minX = fractileFilterImg.size().width - 1;
291         minY = fractileFilterImg.size().height - 1;
292         maxX = maxY = 0;
293         for (size_t y = 0; y < fractileFilterImg.size().height; y++) {
294             for (size_t x = 0; x < fractileFilterImg.size().width; x++) {
295                 if (fractileFilterImg.data[index]) { // Look for all white
                        pixels
296                     if (x < minX)
297                         minX = x;
298                     else if (x > maxX)
299                         maxX = x;
300                     if (y < minY)
301                         minY = y;
302                     else if (y > maxY)
303                         maxY = y;
304                 }
305                 index += fractileFilterImg.channels();
306             }
307         }
308         //printf("%d,%d,%d,%d\n", minX, maxX, minY, maxY);
309
310         if (maxX == 0 || maxY == 0)
311             continue; // Skip, as there was no object detected
312
313         minX -= cropPadding;
314         minY -= cropPadding;
```

```
315         if (minX < 0)
316             minX = 0;
317         if (minY < 0)
318             minY = 0;
319 
320         int width = maxX - minX;
321         int height = maxY - minY;
322         width += cropPadding;
323         height += cropPadding;
324         if (minX + width >= fractileFilterImg.size().width)
325             width = fractileFilterImg.size().width - 1 - minX;
326         if (minY + height >= fractileFilterImg.size().height)
327             height = fractileFilterImg.size().height - 1 - minY;
328 
329         fractileFilterImg = Mat(fractileFilterImg, Rect(minX, minY, width,
                height)).clone(); // Do the actual cropping
330 #else
331         int minX = 0, minY = 0;
332 #endif
333 
334 #if PRINT_TIMING
335         printf("Crop = %f ms\t", ((double)getTickCount() - timer) /
                getTickFrequency() * 1000.0);
336         timer = (double)getTickCount();
337 #endif
338 
339         // Apply morphological closing and opening
340         Mat morphologicalFilterImg = fractileFilterImg.clone();
341         // Morphological closing (Remove small dark spots (i.e. "pepper") and
                connect small bright cracks)
342         morphologicalFilterImg = morphologicalFilter(&morphologicalFilterImg,
                DILATION, closingSize, true);
343         morphologicalFilterImg = morphologicalFilter(&morphologicalFilterImg,
                EROSION, closingSize, true);
344 
345         // Morphological opening (Remove small bright spots (i.e. "salt") and
                connect small dark cracks)
346         morphologicalFilterImg = morphologicalFilter(&morphologicalFilterImg,
                EROSION, openingSize, true);
347         morphologicalFilterImg = morphologicalFilter(&morphologicalFilterImg,
                DILATION, openingSize, true);
348         //imshow("Morphological", morphologicalFilterImg);
349 #if WRITE_IMAGES
350         imwrite("img/morphologicalFilterImg.png", morphologicalFilterImg);
351 #endif
352 
353 #if PRINT_TIMING
354         printf("Morph = %f ms\t", ((double)getTickCount() - timer) /
                getTickFrequency() * 1000.0);
355         timer = (double)getTickCount();
356 #endif
357 
358         // Create a image for each segment
359         uint8_t nSegments;
360         Mat *segments = getSegments(&morphologicalFilterImg, &nSegments,
                neighborSize, true);
361 #if PRINT_TIMING
362         printf("Segments = %f ms\t", ((double)getTickCount() - timer) /
                getTickFrequency() * 1000.0);
363         timer = (double)getTickCount();
364 #endif
365 
366         // Draw red contour if object is found
367         moments_t moments[nSegments];
368         uint8_t objectsDetected = 0;
```

```
369         for (uint8_t i = 0; i < nSegments; i++) {
370             moments_t momentsTmp = calculateMoments(&segments[i], true);
371             int16_t eulerNumber = calculateEulerNumber(&segments[i],
                    CONNECTED_8, true);
372 
373             // Object detected if it is within the range of the invariant,
                    has the right area and Euler number is equal to 1
374             if (momentsTmp.phi1 > (float)objectMin * 1e-4f && momentsTmp.phi1
                     < (float)objectMax * 1e-4f && momentsTmp.area > areaMin &&
                    momentsTmp.area < areaMax && eulerNumber == 1) {
375                 const float sideLength = sqrtf(momentsTmp.area); // Calculate
                        side length from area
376                 const float hypotenuse = sqrtf(2 * sideLength * sideLength);
                        // Calculate hypotenuse, assuming that it is square
377                 //printf("Area: %f %f %f\n", moments.area, sideLength,
                        hypotenuse);
378                 momentsTmp.centerX += minX; // Convert to x,y coordinates in
                        original image
379                 momentsTmp.centerY += minY;
380                 image = drawMoments(&image, &momentsTmp, hypotenuse / 9.0f,
                        0); // Draw center of mass on original image
381                 moments[objectsDetected++] = momentsTmp; // Save the moments
                        of detected objects
382     #if 0 // Use Laplacian filter with lowpass filter to draw the contour
383                 static LinearFilter lowpassLaplacianFilter = LaplacianFilter
                        () + LowpassFilter();
384                 Mat contour = lowpassLaplacianFilter.apply(&segments[i]); //
                        Calculate contour
385     #elif 0 // Use Laplacian filter to draw the contour
386                 static LaplacianFilter laplacianFilter;
387                 Mat contour = laplacianFilter.apply(&segments[i]); //
                        Calculate contour
388     #else // Use contour search method
389                 Mat contour;
390                 if (contoursSearch(&segments[i], &contour, CONNECTED_8, true)
                        ) // When there is only one object in each segment it is
                         faster to use the contours search
391     #endif
392                 {
393                     //imshow("contour", contour);
394 #if WRITE_IMAGES
395                     imwrite("img/contour.png", contour);
396 #endif
397 
398                     index = 0;
399                     for (size_t y = 0; y < contour.size().height; y++) {
400                         for (size_t x = 0; x < contour.size().width; x++) {
401                             if (contour.data[index]) {
402                                 size_t subIndex = ((x + minX) + (y + minY) *
                                        image.size().width) * image.channels();
                                        // Convert to x,y coordinates in
                                        original image
403                                 image.data[subIndex + 0] = 0;
404                                 image.data[subIndex + 1] = 0;
405                                 image.data[subIndex + 2] = 255; // Draw red
                                        contour
406                             }
407                             index++;
408                         }
409                     }
410                 }
411             } /*else if (DEBUG)
412                 printf("Segment: %u\tPhi: %.4f,%.4f\tEuler number: %d\
                        tSegments: %u\n", i, moments.phi1, moments.phi2,
                        eulerNumber, nSegments);*/
```

```
413         }
414 #if PRINT_TIMING
415         printf("Contour = %f ms\t", ((double)getTickCount() - timer) /
                getTickFrequency() * 1000.0);
416 #endif
417 #if WRITE_IMAGES
418         imwrite("img/image_contour.png", image);
419 #endif
420 
421         if (DEBUG) {
422             // Create windows
423             Mat window1(2 * image.size().height, image.size().width, CV_8UC3)
                    ;
424             Mat top1(window1, Rect(0, 0, image.size().width, image.size().
                    height));
425             Mat bot1(window1, Rect(0, image.size().height, imgThresholded.
                    size().width, imgThresholded.size().height));
426 
427             Mat window2(2 * fractileFilterImg.size().height,
                    fractileFilterImg.size().width, CV_8UC3);
428             Mat top2(window2, Rect(0, 0, fractileFilterImg.size().width,
                    fractileFilterImg.size().height));
429             Mat bot2(window2, Rect(0, fractileFilterImg.size().height,
                    morphologicalFilterImg.size().width, morphologicalFilterImg.
                    size().height));
430 
431             // Convert to color images, so it actually shows up
432             cvtColor(imgThresholded, imgThresholded, COLOR_GRAY2BGR);
433             cvtColor(fractileFilterImg, fractileFilterImg, COLOR_GRAY2BGR);
434             cvtColor(morphologicalFilterImg, morphologicalFilterImg,
                    COLOR_GRAY2BGR);
435 
436             // Copy images to windows
437             image.copyTo(top1);
438             imgThresholded.copyTo(bot1);
439             fractileFilterImg.copyTo(top2);
440             morphologicalFilterImg.copyTo(bot2);
441 
442             imshow("Window1", window1);
443             imshow("Window2", window2);
444         }
445 
446 #if __arm__
447         static double zombieDeathTimer = 0;
448         static const uint16_t waitTime = 250; // Wait x ms after solenoid has
                gone all the way up down, as the zombie need to vanish
449         static bool waitForSolenoidDone = false;
450         if (solenoidDone && waitForSolenoidDone) {
451             waitForSolenoidDone = false;
452             zombieDeathTimer = (double)getTickCount(); // Set timer value
453         }
454 
455         static const int8_t topBorder = 5, bottomBorder = 20, middleOffset =
                -10;
456         if (DEBUG || WRITE_IMAGES) {
457                 // Draw blue lines to indicate borders and middle
458                 line(image, Point(0, topBorder), Point(image.size().width,
                        topBorder), Scalar(255, 0, 0)); // Top border
459                 line(image, Point(0, image.size().height / 2 + middleOffset),
                         Point(image.size().width, image.size().height / 2 +
                        middleOffset), Scalar(255, 0, 0)); // Middle
460                 line(image, Point(0, image.size().height - bottomBorder),
                        Point(image.size().width, image.size().height -
                        bottomBorder), Scalar(255, 0, 0)); // Bottom border
461                 if (DEBUG)
```

```
462                     imshow("Areas", image);
463 #if WRITE_IMAGES
464                 imwrite("img/areas.png", image);
465 #endif
466         }
467
468         // Sort moments in ascending order according to the centerX position
469         // Inspired by: http://www.tenouk.com/cpluscodesnippet/
                sortarrayelementasc.html
470         for (uint8_t i = 1; i < objectsDetected; i++) {
471             for (uint8_t j = 0; j < objectsDetected - 1; j++) {
472                 if (moments[j].centerX > moments[j + 1].centerX) {
473                     moments_t temp = moments[j];
474                     moments[j] = moments[j + 1];
475                     moments[j + 1] = temp;
476                 }
477             }
478         }
479
480 #if 0 // Used to analyze the images
481         static uint16_t counter = 0;
482         char buf[30];
483         sprintf(buf, "img/image%u.jpg", counter++);
484         imwrite(buf, image);
485 #endif
486
487         static uint8_t rounds = 0;
488         static uint8_t nZombies; // Track up to this number of zombies
489         if (rounds < 10) {
490             rounds++;
491             nZombies = 2; // Only track two in the beginning because of the
                    text
492         } else
493             nZombies = 4;
494         static int8_t zombieCounter[4];
495
496         for (uint8_t i = 0; i < nZombies; i++) {
497             if (moments[i].centerY > topBorder && moments[i].centerY < image.
                    size().height - bottomBorder) { // Ignore plants in the
                    border
498                 if ((solenoidDone && (((double)getTickCount() -
                        zombieDeathTimer) / getTickFrequency() * 1000.0 >
                        waitTime))) { // If it has been more than x ms since
                        last zombie was killed or the center x position is above
499                     if (moments[i].centerY > image.size().height / 2 +
                            middleOffset) {
500                         zombieCounter[i]++;
501                         /*if (zombieCounter[i] < 0) // We want x times in a
                                row, so reset counter if it is negative
502                             zombieCounter[i] = 0;*/
503                     } else {
504                         zombieCounter[i]--;
505                         /*if (zombieCounter[i] > 0) // We want x times in a
                                row, so reset counter if it is positive
506                             zombieCounter[i] = 0;*/
507                     }
508                     if (DEBUG)
509                         printf("zombieCounter[%u] = %d\n", i, zombieCounter[i
                                ]);
510                 }
511             }
512         }
513
514         uint8_t zombieDeaths = 0;
515         for (uint8_t i = 0; i < nZombies; i++) {
```

```
516             if (abs(zombieCounter[0]) >= 1) { // Check how many times in a row we have seen that zombie
517                 //lastCenterX = moments[0].centerX; // Store center x value
518                 solenoidDone = false;
519                 waitForSolenoidDone = true;
520                 zombieDeaths++;
521                 if (zombieCounter[0] > 0) {
522                     if (DEBUG)
523                         printf("Right\n");
524                     zombieBuffer.write(1); // Indicate to state machine that it should kill a zombie on the right side
525                 } else {
526                     if (DEBUG)
527                         printf("Left\n");
528                     zombieBuffer.write(-1); // Indicate to state machine that it should kill a zombie on the left side
529                 }
530                 for (uint8_t i = 0; i < nZombies - 1; i++)
531                     zombieCounter[i] = zombieCounter[i + 1]; // Move all one down
532             } else
533                 break;
534         }
535         for (int8_t i = nZombies; i > nZombies - zombieDeaths; i--)
536             zombieCounter[i - 1] = 0; // Reset to initial value, as they are moved down
537 #endif
538
539         double dt =  ((double)getTickCount() - startTimer) / getTickFrequency() * 1000.0;
540         int delay = FPS_MS - dt; // Limit to 50 FPS
541         if (delay <= 0) // If the loop has spent more than 50 FPS we will just wait the minimum amount
542             delay = 1; // Wait at least 1 ms, as we need to read the keyboard, button, and update the solenoid state machine
543         while (delay--) {
544 #if __arm__
545             if (cvWaitKey(1) == 27 || !digitalRead(buttonPin)) // End if either ESC or button is pressed
546 #else
547             if (cvWaitKey(1) == 27) // End if ESC is pressed
548 #endif
549                 goto end;
550 #if __arm__
551             runSolenoidStateMachine(); // This state machine control the solenoids without blocking the code
552 #endif
553         }
554
555 #if PRINT_FPS
556         printf("FPS = %.2f\n", 1.0/(((double)getTickCount() - startTimer) / getTickFrequency()));
557 #endif
558
559 #if PRINT_TIMING
560         printf("Total = %f ms\n", ((double)getTickCount() - startTimer) / getTickFrequency() * 1000.0);
561 #endif
562     }
563
564 end:
565     releaseSegments(); // Release all segments inside segmentation.cpp
566 #if __arm__
567     digitalWrite(rightSolenoidPin, HIGH); // Turn both solenoids off
568     digitalWrite(leftSolenoidPin, HIGH);
```

```
569 #endif
570
571     return 0;
572 }
```

Appendix B

Filter code

```
/* Copyright (C) 2015 Kristian Sloth Lauszus. All rights reserved.

 This software may be distributed and modified under the terms of the GNU
 General Public License version 2 (GPL2) as published by the Free Software
 Foundation and appearing in the file GPL2.TXT included in the packaging of
 this file. Please note that GPL2 Section 2[b] requires that all works based
 on this software must also be made publicly available under the terms of
 the GPL2 ("Copyleft").

 Contact information
 -------------------

 Kristian Sloth Lauszus
 Web      :  http://www.lauszus.com
 e-mail   :  lauszus@gmail.com
*/

#include <opencv2/highgui.hpp>
#include <opencv2/imgproc.hpp>

#include "filter.h"
#include "histogram.h"
#include "misc.h"

using namespace cv;

// Multiply all kernel coefficients with a gain
const LinearFilter operator * (const LinearFilter& filter, const float gain)
    {
    return LinearFilter(filter) *= gain;
}

```

```
32 const LinearFilter operator * (const float gain, const LinearFilter& filter)
       {
33     return filter * gain;
34 }
35
36 // Some handy linear filter kernels
37 const float LowpassFilter::lowpass[3 * 3] = {
38     1, 1, 1,
39     1, 1, 1,
40     1, 1, 1,
41 };
42
43 const float HighpassFilter::highpass[3 * 3] = {
44     -1, -1, -1,
45     -1,  8, -1,
46     -1, -1, -1,
47 };
48
49 const float LaplacianFilter::laplacian[3 * 3] = {
50     0,  1,  0,
51     1, -4,  1,
52     0,  1,  0,
53 };
54
55 const float LaplacianTriangularFilter::laplacianTriangular[3 * 3] = {
56     1,   0,  1,
57     0,  -4,  0,
58     1,   0,  1,
59 };
60
61 const float LaplaceGaussianFilter::laplaceGaussian[9 * 9] = {
62     0,  0,  1,   2,    2,   2,  1,  0, 0,
63     0,  1,  5,  10,   12,  10,  5,  1, 0,
64     1,  5, 15,  19,   16,  19, 15,  5, 1,
65     2, 10, 19, -19,  -64, -19, 19, 10, 2,
66     2, 12, 16, -64, -148, -64, 16, 12, 2,
67     2, 10, 19, -19,  -64, -19, 19, 10, 2,
68     1,  5, 15,  19,   16,  19, 15,  5, 1,
69     0,  1,  5,  10,   12,  10,  5,  1, 0,
70     0,  0,  1,   2,    2,   2,  1,  0, 0,
71 };
72
73 Mat LinearFilter::apply(const Mat *q) {
74     const Size size = q->size();
75     const int width = size.width;
76     const int height = size.height;
77     const uint8_t channels = q->channels();
78     const size_t nPixels = q->total() * channels;
79
80     Mat p(size, q->type());
81
82     size_t index = 0;
83     for (size_t y = 0; y < height; y++) {
84         for (size_t x = 0; x < width; x++) {
85             float value[3] = { 0, 0, 0 };
86             for (int8_t k = -n; k <= n; k++) {
87                 // Make sure that we are within the height limit
88                 if ((int32_t)(y + k) < 0)
89                     k = -y;
90                 else if (y + k >= height)
91                     break;
92
93                 for (int8_t l = -m; l <= m; l++) {
94                     // Make sure that we are within the width limit
95                     if ((int32_t)(x + l) < 0)
```

```
96                          l = -x;
97                      else if (x + l >= width)
98                          break;
99
100                     int32_t subIndex = index + (k * width + l) * channels;
101                     bool overflow = subIndex <  0 || subIndex >= nPixels;
102                     if (!overflow) {
103                         for (uint8_t i = 0; i < channels; i++)
104                             value[i] += c[rows * (k + n) + (l + m)] * (float)
                                    q->data[subIndex + i];
105                     } else
106                         printf("\nsubIndex: %d\n", subIndex);
107                     assert(!overflow); // Prevent overflow
108                 }
109             }
110             for (uint8_t i = 0; i < channels; i++)
111                 p.data[index + i] = constrain(value[i], 0, 255); // Constrain
                        data into valid range
112             index += channels; // Increment with number of channels. Same as:
                    index = (x + y * width) * channels;
113         }
114     }
115
116     return p;
117 }
118
119 LinearFilter LinearFilter::combineFilterKernels(const LinearFilter filter1,
        const LinearFilter filter2) {
120     // TODO: Can this be fixed by just using max row or just pad smallest
            kernel?
121     assert(filter1.size == filter2.size); // Has to be equal sizes for now
122
123     const uint8_t size1 = filter1.size;
124     const uint8_t size2 = filter2.size;
125     const uint8_t rows1 = sqrtf(size1);
126     const uint8_t rows2 = sqrtf(size2);
127     const uint8_t outRow =  rows1 + rows2 - 1;
128     const uint8_t outSize = outRow * outRow;
129
130     float c[outSize];
131     for (uint8_t y = 0; y < outRow; y++) {
132         for (uint8_t x = 0; x < outRow; x++) {
133             size_t index = x + y * outRow;
134             float total = 0;
135
136             const uint8_t startI = max(0, y - rows1 + 1);
137             const uint8_t stopI = min((uint8_t)(y + 1), rows1);
138
139             const uint8_t startJ = max(0, x - rows1 + 1);
140             const uint8_t stopJ = min((uint8_t)(x + 1), rows1);
141
142             for (uint8_t i = startI; i < stopI; i++) {
143                 for (uint8_t j = startJ; j < stopJ; j++) {
144                     uint8_t subIndex1 = i * rows1 + j;
145                     uint8_t subIndex2 = (size2 - 1) - x - y * rows2 + (j + i
                            * rows2);
146                     total += filter1.c[subIndex1] * filter2.c[subIndex2];
147                     //printf("Index: %lu %u %u\n", index, subIndex1,
                            subIndex2);
148                 }
149             }
150             //printf("\n");
151             c[index] = total;
152         }
153     }
```

```
154
155     return LinearFilter(c, outSize); // Return new filter
156 }
157
158 static void addRemoveToFromHistogram(histogram_t *histogram, const Mat *image
        , bool add) {
159     const Size size = image->size();
160     const int width = size.width;
161     const int height = size.height;
162     const uint8_t channels = image->channels();
163
164     size_t index = 0;
165     for (size_t y = 0; y < height; y++) {
166         for (uint8_t i = 0; i < channels; i++) {
167             if (add) {
168                 uchar val = image->data[(index + i) + ((width - 1) * channels
                        )]; // Read from the right side of image
169                 histogram->data[val][i]++;
170             } else {
171                 uchar val = image->data[index + i]; // Read from left side of
                        image
172                 if (histogram->data[val][i] > 0)
173                     histogram->data[val][i]--;
174                 else {
175                     printf("Size = %dx%d y = %lu i = %u index = %lu\n", width
                            , height, y, i, index);
176                     assert(histogram->data[val][i]); // This should never
                            happen
177                 }
178             }
179         }
180         index += width * channels; // Go to next row
181     }
182 }
183
184 Mat fractileFilter(const Mat *image, const uint8_t windowSize, const uint8_t
        percentile, bool skipBlackPixels) {
185     const Size size = image->size();
186     const int width = size.width;
187     const int height = size.height;
188     const uint8_t channels = image->channels();
189
190     assert(!skipBlackPixels || (skipBlackPixels && channels == 1)); // If
            skipping black pixels, then the image must be in black and white
191
192     Mat filteredImage(size, image->type());
193     memset(filteredImage.data, 0, filteredImage.total());
194
195     // TODO: Just read directly from image instead of copying data to new
            window
196
197     size_t index = 0;
198     histogram_t histogram;
199     for (size_t y = 0; y < height; y++) {
200         for (size_t x = 0; x < width; x++) {
201             // If the picture is only black and white and looking for white
                    pixels, then it is a good idea to set 'skipBlackPixels' to
                    true, as it will save a lot of time!
202             if (skipBlackPixels && image->data[index] == 0) {
203                 index += channels;
204                 continue;
205             }
206
207             // TODO: Fix problem around border, as it does not go to end
                    position
```

```
208             int32_t windowX = x - windowSize / 2;
209             int32_t windowY = y - windowSize / 2;
210
211             uint8_t croppedX = 0, croppedY = 0;
212             if (windowX < 0) {
213                 croppedX = -windowX;
214                 windowX = 0;
215             } if (windowY < 0) {
216                 croppedY = -windowY;
217                 windowY = 0;
218             }
219
220             size_t spanX = x + windowSize / 2;
221             size_t spanY = y + windowSize / 2;
222
223             uint8_t windowWidth = spanX >= width ? windowSize - (spanX -
                    width) - 1: windowSize;
224             uint8_t windowHeight = spanY >= height ? windowSize - (spanY -
                    height) - 1: windowSize;
225             windowWidth -= croppedX;
226             windowHeight -= croppedY;
227
228             //printf("%lu\t%d,%d\t%lu,%lu\t%u,%u\n", index, windowX, windowY,
                     spanX, spanY, windowWidth, windowHeight);
229
230             // Don't create new window each turn, simply read image directly
231             Mat window = Mat(*image, Rect(windowX, windowY, windowWidth,
                    windowHeight)).clone();
232             /*imshow("Fractile window", window);
233             cvWaitKey(100);*/
234
235             // TODO: Fix this hack
236             //static uint32_t counter = 0;
237             if (!skipBlackPixels && windowWidth == windowSize && windowHeight
                     == windowSize) // We have to recalculate the histogram
                    every time if we are skipping pixels
238                 addRemoveToFromHistogram(&histogram, &window, true); // Add
                        next side to histogram
239             else {
240                 histogram = getHistogram(&window); // Only use this routine
                        the first time
241                 //printf("Counter: %d %u %u %u %u\n", ++counter, windowX,
                        windowY, windowWidth, windowHeight);
242             }
243
244             // Now find median from histogram
245             // TODO: Optimize this
246             const uint32_t medianPos = windowWidth * windowHeight *
                    percentile / 100;
247             uint32_t total[3] = { 0, 0, 0 };
248             int median[3] =  { -1, -1, -1 };
249             for (uint16_t i = 0; i < histogram.nSize; i++) {
250                 for (uint8_t j = 0; j < channels; j++) {
251                     //printf("[%d] = %d\n", i, histogram.data[i][j]);
252                     if (median[j] == -1) { // Only run if median is not found
                            yet
253                         total[j] += histogram.data[i][j];
254                         if (total[j] >= medianPos) {
255                             median[j] = i; // Median found
256
257                             bool done = true;
258                             for (uint8_t k = 0; k < channels; k++) {
259                                 if (median[k] == -1)
260                                     done = false;
261                             }
```

```
262                             if (done) // Break if all medians are found
263                                 break;
264                         }
265                     }
266                 }
267             }
268
269             for (uint8_t i = 0; i < channels; i++) {
270                 //printf("Median[%u]: %d at %u\n", i, median[i], medianPos);
271                 filteredImage.data[index + i] = median[i];
272             }
273             index += channels;
274
275             if (!skipBlackPixels && windowWidth == windowSize && windowHeight
                    == windowSize)
276                 addRemoveToFromHistogram(&histogram, &window, false); //
                        Remove left side of window from histogram
277         }
278     }
279
280     /*histogram_t histogram = getHistogram(&filteredImage);
281     imshow("His", drawHistogram(&histogram, &filteredImage, Size(600, 400)));
            */
282
283     return filteredImage;
284 }
285
286 // TODO: Optimize this, as this is a very slow approach
287 Mat morphologicalFilter(const Mat *image, MorphologicalType type, const
        uint8_t structuringElementSize, bool whitePixels) {
288     assert(image->channels() == 1); // Picture must be a greyscale image
289
290     const Size size = image->size();
291     const int width = size.width;
292     const int height = size.height;
293     const uint8_t n = (structuringElementSize - 1)/2;
294
295     Mat filteredImage = image->clone(); // Create copy of original image
296     size_t index = 0;
297     for (size_t y = 0; y < height; y++) {
298         for (size_t x = 0; x < width; x++) {
299             uint8_t minMax = image->data[index];
300             if ((type == DILATION && whitePixels) || (type == EROSION && !
                    whitePixels)) { // Max is used for dilation when looking for
                     white pixels
301                 if (minMax == 255) // It is already the maximum possible
                        value
302                     goto breakout;
303             } else { // Max is used for erosion when looking for white pixels
304                 if (minMax == 0) // It is already the minimum possible value
305                     goto breakout;
306             }
307             for (int i = -n; i <= n; i++) { // Look around image
308                 for (int j = -n; j <= n; j++) {
309                     size_t subIndex = index + i * width + j;
310                     if (subIndex > 0 && subIndex < width * height && !(i == 0
                            && j == 0)) { // Prevent overflow and make sure
                            that we are not looking at the center pixel again
311                         if ((type == DILATION && whitePixels) || (type ==
                                EROSION && !whitePixels)) { // Max is used for
                                dilation when looking for white pixels
312                             if (image->data[subIndex] > minMax)
313                                 minMax = image->data[subIndex]; // Update
                                        maximum value
314                             if (minMax == 255)
```

```
315                                 goto breakout; // Maximum possible value is
                                        found
316                             } else { // Max is used for erosion when looking for
                                     white pixels
317                                 if (image->data[subIndex] < minMax)
318                                     minMax = image->data[subIndex]; // Update
                                        minimum value
319                                 if (minMax == 0)
320                                     goto breakout; // Minimum possible value is
                                        found
321                             }
322                         }
323                     }
324                 }
325 breakout:
326                 filteredImage.data[index] = minMax; // Set pixel to min/max value
                        of its neighbors
327                 index++; // Increment index
328             }
329         }
330         return filteredImage;
331 }
```

```
1  /* Copyright (C) 2015 Kristian Sloth Lauszus. All rights reserved.
2
3   This software may be distributed and modified under the terms of the GNU
4   General Public License version 2 (GPL2) as published by the Free Software
5   Foundation and appearing in the file GPL2.TXT included in the packaging of
6   this file. Please note that GPL2 Section 2[b] requires that all works based
7   on this software must also be made publicly available under the terms of
8   the GPL2 ("Copyleft").
9
10  Contact information
11  -------------------
12
13  Kristian Sloth Lauszus
14  Web      :  http://www.lauszus.com
15  e-mail   :  lauszus@gmail.com
16 */
17
18 #ifndef __filter_h__
19 #define __filter_h__
20
21 using namespace cv;
22
23 enum MorphologicalType {
24     EROSION = 0,
25     DILATION,
26 };
27
28 Mat fractileFilter(const Mat *image, const uint8_t windowSize, const uint8_t
       percentile, bool skipBlackPixels);
29 Mat morphologicalFilter(const Mat *image, MorphologicalType type, const
       uint8_t structuringElementSize, bool whitePixels);
30
31 class LinearFilter {
32 public:
33     // Assumes that the kernel is symmetric
34     LinearFilter(const float *coefficients, uint8_t _size) :
35         n(-0.5f + 0.5f * sqrtf(_size)),
36         m(n),
37         rows(2 * n + 1),
38         columns(2 * m + 1),
```

```
39          size(_size),
40          lastSum(0) {
41          initCoefficients(coefficients, true);
42      }
43
44      // Set kernel using n and m
45      LinearFilter(const float *coefficients, const uint8_t _n, const uint8_t
            _m) :
46          n(_n),
47          m(_m),
48          rows(2 * n + 1),
49          columns(2 * m + 1),
50          size(rows * columns),
51          lastSum(0) {
52          initCoefficients(coefficients, true);
53      }
54
55      // Copy constructor
56      LinearFilter(const LinearFilter& filter) {
57          swap(filter);
58      }
59
60      ~LinearFilter() {
61          delete[] c;
62      }
63
64      inline Mat operator () (const Mat *q) {
65          return apply(q); // Apply filter
66      }
67
68      // Assignment operator
69      LinearFilter& operator = (const LinearFilter& filter) {
70         if (this != &filter)
71             swap(filter);
72         return *this;
73     }
74
75      // Used to combine two filter kernels into one
76      const LinearFilter operator + (const LinearFilter& filter) {
77          return *this += filter;
78      }
79
80      LinearFilter& operator += (const LinearFilter& filter) {
81          if (lastSum != 0) { // Make sure that it has been normalized
82              for (uint8_t i = 0; i < size; i++)
83                  c[i] *= lastSum; // Undo normalization
84          }
85          if (filter.lastSum != 0) { // Make sure that it has been normalized
86              for (uint8_t i = 0; i < size; i++)
87                  c[i] *= filter.lastSum; // Undo normalization. We can just
                        multiply the current filter with the other gain, as it
                        has the same effect in the end
88          }
89          return *this = combineFilterKernels(*this, filter);
90      }
91
92      // Multiply all kernel coefficients with a gain
93      LinearFilter& operator *= (const float gain) {
94          for (uint8_t i = 0; i < size; i++)
95              c[i] *= gain;
96          if (gain != 0) { // Prevent division with zero
97              if (lastSum == 0)
98                  lastSum = 1; // If it has not been set before, set it equal
                        to the inverse of the gain
99              lastSum /= gain; // Update last sum, so it can undo normalization
```

```
                correctly
        }
        return *this;
    }

    Mat apply(const Mat *q);

    void printKernel(void) {
        for (uint8_t i = 0; i < rows; i++) {
            for (uint8_t j = 0; j < columns; j++)
                printf("%.2f\t", c[i * rows + j]);
            printf("\n");
        }
    }

    uint8_t n, m;
    uint8_t rows, columns;
    uint8_t size;
    float *c;

private:
    void swap(const LinearFilter& filter) {
        n = filter.n;
        m = filter.m;
        rows = filter.rows;
        columns = filter.columns;
        size = filter.size;
        initCoefficients(filter.c, false); // Don't normalize, just copy data
    }

    void initCoefficients(const float *coefficients, bool _normalize) {
        //printf("n = %u m = %u rows = %u columns = %u size = %u normalize  =
            %u\n", n, m, rows, columns, size, _normalize);

        c = new float[size];

        float sum = 0;
        for (uint8_t i = 0; i < size; i++) {
            c[i] = coefficients[i];
            sum += c[i];
        }
        if (_normalize && sum != 0 && sum != 1)
            normalize();
    }

    void normalize(void) {
        float sum = 0;
        for (uint8_t i = 0; i < size; i++)
            sum += c[i];

        float sum_new = 0;
        if (sum < -0.0001f || sum > 0.0001f) { // Skip if value is very close
             to 0
            for (uint8_t i = 0; i < size; i++) {
                c[i] /= sum; // Normalize data
                sum_new += c[i];
            }
            //printf("Normalized: %f %.2f\n", sum, sum_new);
            assert(sum_new == 1.00f);
            lastSum = sum;
        }
    }

    LinearFilter combineFilterKernels(const LinearFilter filter1, const
        LinearFilter filter2);
```

```
161
162     float lastSum; // Used to undo normalization
163 };
164
165 // Multiply filter all kernel coefficients with a gain
166 const LinearFilter operator * (const LinearFilter& filter, const float gain);
167 const LinearFilter operator * (const float gain, const LinearFilter& filter);
168
169 class LowpassFilter : public LinearFilter {
170 public:
171     LowpassFilter() :
172         LinearFilter(lowpass, sizeof(lowpass)/sizeof(lowpass[0])) {
173     }
174
175     static const float lowpass[3 * 3];
176 };
177
178 class HighpassFilter : public LinearFilter {
179 public:
180     HighpassFilter() :
181         LinearFilter(highpass, sizeof(highpass)/sizeof(highpass[0])) {
182     }
183
184     static const float highpass[3 * 3];
185 };
186
187 class LaplacianFilter : public LinearFilter {
188 public:
189     LaplacianFilter() :
190         LinearFilter(laplacian, sizeof(laplacian)/sizeof(laplacian[0])) {
191     }
192
193     static const float laplacian[3 * 3];
194 };
195
196 class LaplacianTriangularFilter : public LinearFilter {
197 public:
198     LaplacianTriangularFilter() :
199         LinearFilter(laplacianTriangular, sizeof(laplacianTriangular)/sizeof(
                laplacianTriangular[0])) {
200     }
201
202     static const float laplacianTriangular[3 * 3];
203 };
204
205 class LaplaceGaussianFilter : public LinearFilter {
206 public:
207     LaplaceGaussianFilter() :
208         LinearFilter(laplaceGaussian, sizeof(laplaceGaussian)/sizeof(
                laplaceGaussian[0])) {
209     }
210
211     static const float laplaceGaussian[9 * 9];
212 };
213
214 #endif
```

Appendix C

Segmentation code

```
/* Copyright (C) 2015 Kristian Sloth Lauszus. All rights reserved.

 This software may be distributed and modified under the terms of the GNU
 General Public License version 2 (GPL2) as published by the Free Software
 Foundation and appearing in the file GPL2.TXT included in the packaging of
 this file. Please note that GPL2 Section 2[b] requires that all works based
 on this software must also be made publicly available under the terms of
 the GPL2 ("Copyleft").

 Contact information
 -------------------

 Kristian Sloth Lauszus
 Web      :  http://www.lauszus.com
 e-mail   :  lauszus@gmail.com
*/

#include <opencv2/highgui.hpp>
#include <opencv2/imgproc.hpp>

#include "segmentation.h"

using namespace cv;

static const uint8_t MAX_SEGMENTS = 10;
static Mat segmentImg[MAX_SEGMENTS];

static uint8_t checkNeighbours(const Mat *image, const Mat *segments, const
    size_t index, const int8_t neighbourSize, bool whitePixels) {
    const Size size = image->size();
    const int width = size.width;
    const size_t total = image->total();
```

```
32
33      uint8_t id = 0;
34
35      // Check all neighbors using 8-connectedness
36      for (int8_t i = -neighbourSize; i <= neighbourSize; i++) {
37          for (int8_t j = -neighbourSize; j <= neighbourSize; j++) {
38              int32_t subIndex = index + i * width + j;
39              if (!(i == 0 && j == 0) && subIndex >= 0 && subIndex < total) {
                    // Do not check x,y = (0,0) and prevent overflow
40                  if ((bool)image->data[subIndex] == whitePixels) {
41                      if (segments->data[subIndex] > 0) // Part of an existing
                            segment
42                          return segments->data[subIndex]; // Return ID
43                      else
44                          id = 255; // Part of a new segments, but keep looping
                                as there might be a neighbor with an existing
                                ID
45                  }
46              }
47          }
48      }
49      return id;
50  }
51
52  Mat *getSegments(const Mat *image, uint8_t *nSegments, const int8_t
        neighbourSize, bool whitePixels) {
53      assert(image->channels() == 1); // Picture must be a binary image
54
55      const Size size = image->size();
56      const int width = size.width;
57      const int height = size.height;
58
59      Mat segments(size, image->type()); // This Mat variable is used to store
            the ID of the different segments
60      memset(segments.data, 0, image->total());
61
62      // Look for segments
63      size_t index = 0;
64      *nSegments = 0;
65      for (size_t y = 0; y < height; y++) {
66          for (size_t x = 0; x < width; x++) {
67              if ((bool)image->data[index] == whitePixels) { // Check if we
                    have found a pixel
68                  uint8_t id = checkNeighbours(image, &segments, index,
                        neighbourSize, whitePixels);
69                  if (id > 0) {
70                      if (id == 255) // New segment
71                          id = ++(*nSegments); // Set ID equal to the current
                                number of found segments
72                      segments.data[index] = id; // Set same ID as neighbor
73                  } else
74                      segments.data[index] = 0; // No neighbors, just assume it
                            is noise
75              }
76              index++; // Increment index
77          }
78      }
79
80      // Create an image for each segment
81      if (*nSegments > 0) {
82          if (*nSegments > MAX_SEGMENTS) {
83              *nSegments = MAX_SEGMENTS; // Limit number of segments
84              printf("Segment saturation!\n");
85          }
86
```

```
87          for (uint8_t i=0; i < *nSegments; i++) {
88              segmentImg[i].create(size, segments.type()); // Recreate image
89              memset(segmentImg[i].data, 0, segmentImg[i].total()); // Reset
                    all data
90          }
91
92          for (uint8_t id = 1; id <= *nSegments; id++) {
93              for (size_t j = 0; j < width * height; j++) {
94                  if (segments.data[j] == id)
95                      segmentImg[id - 1].data[j] = 255; // Draw white on each
                            segment
96              }
97  #if 0
98              char buf[20];
99              sprintf(buf, "Segment %u", id);
100             imshow(buf, segmentImg[id - 1]);
101 #endif
102         }
103
104         //printf("Segments: %u\n", *nSegments);
105         return segmentImg;
106     }
107     return NULL;
108 }
109
110 void releaseSegments(void) {
111     for (uint8_t i = 0; i < MAX_SEGMENTS; i++)
112         segmentImg[i].release();
113 }
```

```
1  /* Copyright (C) 2015 Kristian Sloth Lauszus. All rights reserved.
2
3   This software may be distributed and modified under the terms of the GNU
4   General Public License version 2 (GPL2) as published by the Free Software
5   Foundation and appearing in the file GPL2.TXT included in the packaging of
6   this file. Please note that GPL2 Section 2[b] requires that all works based
7   on this software must also be made publicly available under the terms of
8   the GPL2 ("Copyleft").
9
10  Contact information
11  -------------------
12
13  Kristian Sloth Lauszus
14  Web      :  http://www.lauszus.com
15  e-mail   :  lauszus@gmail.com
16 */
17
18 #ifndef __segmentation_h__
19 #define __segmentation_h__
20
21 using namespace cv;
22
23 Mat *getSegments(const Mat *image, uint8_t *nSegments, const int8_t
       neightborSize, bool whitePixels);
24 void releaseSegments(void);
25
26 #endif
```

Appendix D

Contours code

```
/* Copyright (C) 2015 Kristian Sloth Lauszus. All rights reserved.

 This software may be distributed and modified under the terms of the GNU
 General Public License version 2 (GPL2) as published by the Free Software
 Foundation and appearing in the file GPL2.TXT included in the packaging of
 this file. Please note that GPL2 Section 2[b] requires that all works based
 on this software must also be made publicly available under the terms of
 the GPL2 ("Copyleft").

 Contact information
 -------------------

 Kristian Sloth Lauszus
 Web      :  http://www.lauszus.com
 e-mail   :  lauszus@gmail.com
*/

#include <opencv2/imgproc.hpp>

#include "contours.h"
#include "histogram.h"
#include "misc.h"

using namespace cv;

static void checkNeighbors(const Mat *image, int *pos, uint8_t *index, const
    int width, Connected connected, bool whitePixels) {
    const int contourArray[8] = {
            1,          // (x + 1, y)
            width + 1,  // (x + 1, y + 1)
            width,      // (x, y + 1)
            width - 1,  // (x - 1, y + 1)
```

```
32              -1,          // (x - 1, y)
33              -width - 1, // (x - 1, y - 1)
34              -width,      // (x, y - 1)
35              -width + 1, // (x + 1, y - 1)
36      };
37      *index = (*index + 6) % 8; // Select next search direction. Add 6, so we
            look just above the last pixel
38
39      for (uint8_t i = *index; i < *index + 8;) { // Check 8 pixels around
40          int32_t subIndex = *pos + contourArray[i % 8];
41          if (subIndex >= 0 && subIndex < image->total()) {
42              if ((bool)image->data[subIndex] == whitePixels) {
43                  *pos = subIndex;
44                  *index = i % 8;
45                  return;
46              }
47          }
48          if (connected == CONNECTED_8)
49              i++;
50          else if (connected == CONNECTED_4)
51              i += 2; // Skip all corners
52          else { // When using 6-connected, I use the definition here: https://
                en.wikipedia.org/wiki/Pixel_connectivity#6-connected
53              if (i % 8 != 2 && i % 8 != 6)
54                  i++;
55              else
56                  i += 2; // Skip upper right and lower left corners
57          }
58      }
59  }
60
61  bool contoursSearch(const Mat *image, Mat *out, Connected connected, bool
        whitePixels) {
62      assert(image->channels() == 1); // Image has to be one channel only
63
64      *out = Mat(image->size(), image->type());
65      memset(out->data, 0, out->total());
66
67      const size_t total = out->total();
68      const size_t MAX_RAND = total;
69      const Size size = out->size();
70      const int width = size.width;
71      const int height = size.height;
72
73      int pos = -1;
74      for (size_t y = 0; y < height; y++) {
75          for (size_t x = 0; x < width; x++) {
76              size_t index = x + y * width;
77              if ((bool)image->data[index] == whitePixels) {
78                  pos = index;
79                  goto contourFound;
80              }
81          }
82      }
83      if (pos == -1) {
84          printf("No contour detected!\n");
85          return false;
86      }
87
88  contourFound:
89      size_t count = 0;
90      int newpos = pos;
91      uint8_t draw_type = 0;
92
93      while (1) {
```

```
94          out->data[newpos] = 255; // Draw contour
95  #if 1
96          checkNeighbors(image, &newpos, &draw_type, width, connected,
                whitePixels);
97  #else
98          draw_type = (draw_type + 6) % 8; // Select next search direction
99
100         switch (draw_type) {
101         case 0:
102             if ((bool)image->data[newpos + 1] == whitePixels) {
103                 newpos += 1;
104                 draw_type = 0;
105                 break;
106             }
107             // Intentional fall through
108         case 1:
109             if ((bool)image->data[newpos + width + 1] == whitePixels) {
110                 newpos += width + 1;
111                 draw_type = 1;
112                 break;
113             }
114             // Intentional fall through
115         case 2:
116             if ((bool)image->data[newpos + width] == whitePixels) {
117                 newpos += width;
118                 draw_type = 2;
119                 break;
120             }
121             // Intentional fall through
122         case 3:
123             if ((bool)image->data[newpos + width - 1] == whitePixels) {
124                 newpos += width - 1;
125                 draw_type = 3;
126                 break;
127             }
128             // Intentional fall through
129         case 4:
130             if ((bool)image->data[newpos - 1] == whitePixels) {
131                 newpos -= 1;
132                 draw_type = 4;
133                 break;
134             }
135             // Intentional fall through
136         case 5:
137             if ((bool)image->data[newpos - width - 1] == whitePixels) {
138                 newpos -= width + 1;
139                 draw_type = 5;
140                 break;
141             }
142             // Intentional fall through
143         case 6:
144             if ((bool)image->data[newpos - width] == whitePixels) {
145                 newpos -= width;
146                 draw_type = 6;
147                 break;
148             }
149             // Intentional fall through
150         case 7:
151             if ((bool)image->data[newpos - width + 1] == whitePixels) {
152                 newpos -= width - 1;
153                 draw_type = 7;
154                 break;
155             }
156             // Intentional fall through
157         case 8:
```

```
158             if ((bool)image->data[newpos + 1] == whitePixels) {
159                 newpos += 1;
160                 draw_type = 0;
161                 break;
162             }
163             // Intentional fall through
164         case 9:
165             if ((bool)image->data[newpos + width + 1] == whitePixels) {
166                 newpos += width + 1;
167                 draw_type = 1;
168                 break;
169             }
170             // Intentional fall through
171         case 10:
172             if ((bool)image->data[newpos + width] == whitePixels) {
173                 newpos += width;
174                 draw_type = 2;
175                 break;
176             }
177             // Intentional fall through
178         case 11:
179             if ((bool)image->data[newpos + width - 1] == whitePixels) {
180                 newpos += width - 1;
181                 draw_type = 3;
182                 break;
183             }
184             // Intentional fall through
185         case 12:
186             if ((bool)image->data[newpos - 1] == whitePixels) {
187                 newpos -= 1;
188                 draw_type = 4;
189                 break;
190             }
191             // Intentional fall through
192         case 13:
193             if ((bool)image->data[newpos - width - 1] == whitePixels) {
194                 newpos -= width + 1;
195                 draw_type = 5;
196                 break;
197             }
198             // Intentional fall through
199         case 14:
200             if ((bool)image->data[newpos - width] == whitePixels) {
201                 newpos -= width;
202                 draw_type = 6;
203                 break;
204             }
205         }
206 #endif
207         if (newpos == pos) { // If we are back at the beginning, we declare
                success
208             //printf("Count: %lu\n", count);
209             return true;
210         } else if (++count >= MAX_RAND) { // Abort if the contour is too
                complex
211             printf("Contour is too complex!\n");
212             break;
213         }
214     }
215     printf("Count: %lu\n", count);
216     return false;
217 }
```

```
/* Copyright (C) 2015 Kristian Sloth Lauszus. All rights reserved.

 This software may be distributed and modified under the terms of the GNU
 General Public License version 2 (GPL2) as published by the Free Software
 Foundation and appearing in the file GPL2.TXT included in the packaging of
 this file. Please note that GPL2 Section 2[b] requires that all works based
 on this software must also be made publicly available under the terms of
 the GPL2 ("Copyleft").

 Contact information
 -------------------

 Kristian Sloth Lauszus
 Web      :  http://www.lauszus.com
 e-mail   :  lauszus@gmail.com
*/

#ifndef __contours_h__
#define __contours_h__

#include "misc.h"

using namespace cv;

bool contoursSearch(const Mat *image, Mat *out, Connected connected, bool
    whitePixels);

#endif
```

Appendix E

Euler code

```
/* Copyright (C) 2015 Kristian Sloth Lauszus. All rights reserved.

 This software may be distributed and modified under the terms of the GNU
 General Public License version 2 (GPL2) as published by the Free Software
 Foundation and appearing in the file GPL2.TXT included in the packaging of
 this file. Please note that GPL2 Section 2[b] requires that all works based
 on this software must also be made publicly available under the terms of
 the GPL2 ("Copyleft").

 Contact information
 -------------------

 Kristian Sloth Lauszus
 Web      :  http://www.lauszus.com
 e-mail   :  lauszus@gmail.com
*/

#include <opencv2/imgproc.hpp>

#include "euler.h"

using namespace cv;

int16_t calculateEulerNumber(const Mat *image, const Connected connected,
    bool whitePixels) {
    assert(image->channels() == 1); // Image must be in black and white

    const Size size = image->size();
    const int width = size.width;
    const int height = size.height;

    int16_t eulerNumber;
```

```
32
33      size_t nQ1 = 0, nQ3 = 0, nQD = 0;
34
35      size_t index = 0;
36      for (size_t y = 0; y < height; y++) {
37          for (size_t x = 0; x < width; x++) {
38              uint8_t pixelCounter = 0;
39              int8_t pixelLocation[2];
40              for (uint8_t i = 0; i < 2; i++) { // Run through the 4x4 bit quad
41                  for (uint8_t j = 0; j < 2; j++) {
42                      size_t subIndex = index + i * width + j;
43                      if (subIndex < width * height) { // Prevent overflow in
                             the bottom of the image
44                          if ((bool)image->data[subIndex] == whitePixels) {
45                              if (pixelCounter < 2)
46                                  pixelLocation[pixelCounter] = i + 2 * j; //
                                         0,0 = 0; 1,0 = 1; 0,1 = 2; 1,1 = 3
47                              pixelCounter++;
48                          }
49                      }
50                  }
51              }
52              if (pixelCounter == 1)
53                  nQ1++; // There is only 1 pixel in the bit quad
54              else if (pixelCounter == 3)
55                  nQ3++; // There is 3 pixels in the bit quad
56              else if (pixelCounter == 2) {
57                  if ((pixelLocation[0] == 0 && pixelLocation[1] == 3) || (
                         pixelLocation[0] == 1 && pixelLocation[1] == 2))
58                      nQD++; // The bit quad is diagonal
59              }
60              index++; // Increment index
61          }
62      }
63
64      if (connected == CONNECTED_4) // 4-connected
65          eulerNumber = (nQ1 - nQ3 + 2 * nQD) / 4;
66      else // 8-connected
67          eulerNumber = (nQ1 - nQ3 - 2 * nQD) / 4;
68
69      return eulerNumber;
70  }
```

```
1   /* Copyright (C) 2015 Kristian Sloth Lauszus. All rights reserved.
2
3    This software may be distributed and modified under the terms of the GNU
4    General Public License version 2 (GPL2) as published by the Free Software
5    Foundation and appearing in the file GPL2.TXT included in the packaging of
6    this file. Please note that GPL2 Section 2[b] requires that all works based
7    on this software must also be made publicly available under the terms of
8    the GPL2 ("Copyleft").
9
10   Contact information
11   -------------------
12
13   Kristian Sloth Lauszus
14   Web      :  http://www.lauszus.com
15   e-mail   :  lauszus@gmail.com
16  */
17
18  #ifndef __euler_h__
19  #define __euler_h__
20
```

```
21 #include "misc.h"
22
23 using namespace cv;
24
25 int16_t calculateEulerNumber(const Mat *image, const Connected connected,
       bool whitePixels);
26
27 #endif
```

Appendix F

Moments code

```
/* Copyright (C) 2015 Kristian Sloth Lauszus. All rights reserved.

 This software may be distributed and modified under the terms of the GNU
 General Public License version 2 (GPL2) as published by the Free Software
 Foundation and appearing in the file GPL2.TXT included in the packaging of
 this file. Please note that GPL2 Section 2[b] requires that all works based
 on this software must also be made publicly available under the terms of
 the GPL2 ("Copyleft").

 Contact information
 -------------------

 Kristian Sloth Lauszus
 Web      :  http://www.lauszus.com
 e-mail   :  lauszus@gmail.com
*/

#include <opencv2/imgproc.hpp>

#include "moments.h"

using namespace cv;

moments_t calculateMoments(const Mat *image, bool whitePixels) {
    assert(image->channels() == 1); // Picture must be black and white image

    moments_t moments;
    // Calculate moments
    moments.M00 = moments.M10 = moments.M01 = moments.M11 = moments.M20 =
        moments.M02 = 0;
    size_t index = 0;
    for (int y = 0; y < image->size().height; y++) {
```

```
32          for (int x  = 0; x < image->size().width; x++) {
33              if ((bool)image->data[index++] == whitePixels) {
34                  moments.M00++;
35                  // First moments
36                  moments.M10 += x;
37                  moments.M01 += y;
38                  // Second moments
39                  moments.M11 += x * y;
40                  moments.M20 += x * x;
41                  moments.M02 += y * y;
42              }
43          }
44      }
45
46      // Calculate the center of mass
47      moments.centerX = moments.M10 / moments.M00;
48      moments.centerY = moments.M01 / moments.M00;
49
50      // Calculate reduced central moments
51      moments.u00 = moments.M00;
52  #if 1
53      moments.u11 = moments.M11 - moments.centerY * moments.M10;
54  #if 0
55      const float u11_x = moments.M11 - moments.centerX * moments.M01;
56      //printf("%.2f == %.2f\n", moments.u11, u11_x); // Should be the same
57      assert(moments.u11 == u11_x);
58  #endif
59      moments.u20 = moments.M20 - moments.centerX * moments.M10;
60      moments.u02 = moments.M02 - moments.centerY * moments.M01;
61  #else
62      moments.u11 = moments.u20 = moments.u02 = 0;
63      index = 0;
64      for (int y = 0; y < image->size().height; y++) {
65              for (int x  = 0; x < image->size().width; x++) {
66                  if ((bool)image->data[index++] == whitePixels) {
67                      moments.u11 += (x - moments.centerX) * (y - moments.
                            centerY);
68                      moments.u20 += (x - moments.centerX) * (x - moments.
                            centerX);
69                      moments.u02 += (y - moments.centerY) * (y - moments.
                            centerY);
70                  }
71              }
72      }
73  #endif
74
75  #if 0 // Set to 1 in order to normalize data
76      moments.u11 /= moments.u00;
77      moments.u20 /= moments.u00;
78      moments.u02 /= moments.u00;
79  #endif
80
81      // Calculate angle
82      moments.angle = 0.5f * atan2f(2.0f * moments.u11, moments.u20 - moments.
            u02);
83      //printf("Center: %.2f,%.2f\tAngle: %.2f\n", moments.centerX, moments.
            centerY, moments.angle * 180.0f / M_PI);
84
85      // Normalized central moments
86      moments.n11 = moments.u11 / (moments.u00 * moments.u00); // Gamma = (1 +
            1) / 2 + 1 = 2
87      moments.n20 = moments.u20 / (moments.u00 * moments.u00); // Gamma = (2 +
            0) / 2 + 1 = 2
88      moments.n02 = moments.u02 / (moments.u00 * moments.u00); // Gamma = (2 +
            0) / 2 + 1 = 2
```

```
89
90      // Invariant moments
91      moments.phi1 = moments.n20 + moments.n02;
92      moments.phi2 = (moments.n20 + moments.n02) * (moments.n20 + moments.n02)
            + 4 * (moments.n11 * moments.n11);
93      //printf("n: %.2f,%.2f\tphi: %.2f,%.2f\n", moments.n20, moments.n02,
            moments.phi1, moments.phi2);
94
95      return moments;
96  }
97
98  Mat drawMoments(const Mat *image, moments_t *moments, const float
        centerLength, const float angleLineLength) {
99      Mat out;
100     if (image->channels() == 1)
101         cvtColor(*image, out, COLOR_GRAY2BGR); // Convert image into 3-
                channel image, so we can draw in color
102     else
103         out = image->clone();
104
105     // Draw center of mass if length is larger than 0
106     if (centerLength > 0) {
107         line(out, Point(moments->centerX - centerLength, moments->centerY -
                centerLength), Point(moments->centerX + centerLength, moments->
                centerY + centerLength), Scalar(0, 0, 255));
108         line(out, Point(moments->centerX - centerLength, moments->centerY +
                centerLength), Point(moments->centerX + centerLength, moments->
                centerY - centerLength), Scalar(0, 0, 255));
109     }
110
111     // Draw angle line if length is larger than 0
112     if (angleLineLength > 0)
113         line(out, Point(moments->centerX + angleLineLength * cosf(moments->
                angle), moments->centerY + angleLineLength * sinf(moments->angle
                )), Point(moments->centerX - angleLineLength * cosf(moments->
                angle), moments->centerY - angleLineLength * sinf(moments->angle
                )), Scalar(0, 0, 255));
114
115     return out;
116 }
```

```
1   /* Copyright (C) 2015 Kristian Sloth Lauszus. All rights reserved.
2
3    This software may be distributed and modified under the terms of the GNU
4    General Public License version 2 (GPL2) as published by the Free Software
5    Foundation and appearing in the file GPL2.TXT included in the packaging of
6    this file. Please note that GPL2 Section 2[b] requires that all works based
7    on this software must also be made publicly available under the terms of
8    the GPL2 ("Copyleft").
9
10   Contact information
11   -------------------
12
13   Kristian Sloth Lauszus
14   Web      :  http://www.lauszus.com
15   e-mail   :  lauszus@gmail.com
16  */
17
18  #ifndef __moments_h__
19  #define __moments_h__
20
21  using namespace cv;
22
```

```
23 typedef struct moments_t {
24     union {
25         float M00;
26         float area;
27     };
28     float M10, M01, M11, M20, M02; // Moments
29     float centerX, centerY; // Center of mass
30     float u00, u11, u20, u02; // Reduced central moments
31     float angle; // Angle of the object
32     float n11, n20, n02; // Normalized central moments
33     float phi1, phi2; // Invariant moments
34 } moments_t;
35
36 moments_t calculateMoments(const Mat *image, bool whitePixels);
37 Mat drawMoments(const Mat *image, moments_t *moments, const float
       centerLength, const float angleLineLength);
38
39 #endif
```

Appendix G

Histogram code

```
/* Copyright (C) 2015 Kristian Sloth Lauszus. All rights reserved.

 This software may be distributed and modified under the terms of the GNU
 General Public License version 2 (GPL2) as published by the Free Software
 Foundation and appearing in the file GPL2.TXT included in the packaging of
 this file. Please note that GPL2 Section 2[b] requires that all works based
 on this software must also be made publicly available under the terms of
 the GPL2 ("Copyleft").

 Contact information
 -------------------

 Kristian Sloth Lauszus
 Web      :  http://www.lauszus.com
 e-mail   :  lauszus@gmail.com
*/

#include <opencv2/imgproc.hpp>

#include "histogram.h"
#include "misc.h"

using namespace cv;

static int map(int x, int in_min, int in_max, int out_min, int out_max) {
    int value = (x - in_min) * (out_max - out_min) / (in_max - in_min) +
        out_min; // From Arduino source code: https://github.com/arduino/
        Arduino/blob/ide-1.5.x/hardware/arduino/avr/cores/arduino/WMath.cpp
    return constrain(value, out_min, out_max); // Limit output
}

void printHistogram(const histogram_t *histogram, uint8_t channels) {
```

```
31      for (uint16_t i = 0; i < histogram->nSize; i++) {
32          for (uint8_t j = 0; j < channels; j++) {
33              if (histogram->data[i][j])
34                  printf("[%u][%u] %u\n", i, j, histogram->data[i][j]);
35          }
36      }
37  }
38
39  histogram_t getHistogram(const Mat *image) {
40      histogram_t histogram;
41      const uint8_t channels = image->channels();
42
43      size_t index = 0;
44      for (size_t i = 0; i < image->total(); i++) {
45          for (uint8_t j = 0; j < channels; j++) {
46              uchar value = image->data[index + j];
47              histogram.data[value][j]++;
48          }
49          index += channels;
50      }
51
52  #if 0
53      uint32_t total = 0;
54      for (uint16_t i = 0; i < histogram.nSize; i++) {
55          for (uint8_t j = 0; j < channels; j++) {
56              //printf("[%u][%u] = %d\n", i, j, histogram.data[i][j]);
57              total += histogram.data[i][j];
58          }
59      }
60      printf("%u == %lu\n", total, image->total() * channels); // Should be
            equal to "image.total() * image.channels()"
61      assert(total == image->total() * channels);
62  #endif
63
64      return histogram;
65  }
66
67  Mat drawHistogram(const histogram_t *histogram, const Mat *image, const Size
        imageSize, int thresholdValue /*= -1*/) {
68      Mat hist(imageSize, CV_8UC3);
69      rectangle(hist, Point(0, 0), hist.size(), Scalar(255, 255, 255, 0),
            CV_FILLED); // White background
70
71      static const uint8_t offset = 50;
72      static const uint8_t startX = offset;
73      static const uint8_t startY = offset;
74      const int endX = hist.size().width - offset;
75      const int endY = hist.size().height - offset;
76
77      line(hist, Point(startX, endY), Point(endX, endY), Scalar(0)); // x-axis
78      line(hist, Point(startX, startY), Point(startX, endY), Scalar(0)); // y-
            axis
79
80      int maxHist = 0;
81      for (uint16_t i = 0; i < histogram->nSize; i++) {
82          for (uint8_t j = 0; j < image->channels(); j++) {
83              if (histogram->data[i][j] > maxHist)
84                  maxHist = histogram->data[i][j];
85          }
86      }
87
88      for (uint16_t i = 0; i < histogram->nSize; i++) {
89          int posX = map(i, 0, histogram->nSize, startX + 5, endX - 5);
90
91          if (i == thresholdValue)
```

```
92              line(hist, Point(posX, startY), Point(posX, endY), Scalar(0, 0,
                    255, 0)); // Draw red threshold line
93
94          for (uint8_t j = 0; j < image->channels(); j++)
95              line(hist, Point(posX, endY - map(histogram->data[i][j], 0,
                    maxHist, 0, endY - offset)), Point(posX, endY),
96                                              /* Draw in colors if color image,
                                                    else draw black if
                                                   greyscale */
97                                              Scalar(j == 0 && image->channels
                                                    () > 1 ? 255 : 0,
98                                                   j == 1 ? 255 : 0,
99                                                   j == 2 ? 255 : 0, 0));
100     }
101     return hist;
102 }
```

```
1  /* Copyright (C) 2015 Kristian Sloth Lauszus. All rights reserved.
2
3   This software may be distributed and modified under the terms of the GNU
4   General Public License version 2 (GPL2) as published by the Free Software
5   Foundation and appearing in the file GPL2.TXT included in the packaging of
6   this file. Please note that GPL2 Section 2[b] requires that all works based
7   on this software must also be made publicly available under the terms of
8   the GPL2 ("Copyleft").
9
10  Contact information
11  -------------------
12
13  Kristian Sloth Lauszus
14  Web      :  http://www.lauszus.com
15  e-mail   :  lauszus@gmail.com
16 */
17
18 #ifndef __histogram_h__
19 #define __histogram_h__
20
21 using namespace cv;
22
23 struct histogram_t {
24     histogram_t(void) {
25         memset(data, 0, sizeof(data)); // Make sure all elements are zero
26     }
27     static const uint16_t nSize = 256;
28     uint32_t data[nSize][3]; // Data for three channels
29 };
30
31 void printHistogram(const histogram_t *histogram, uint8_t channels);
32 histogram_t getHistogram(const Mat *image);
33 Mat drawHistogram(const histogram_t *histogram, const Mat *image, const Size
       imageSize, int thresholdValue = -1);
34
35 #endif
```

Appendix H

Ringbuffer code

```
/* Copyright (C) 2015 Kristian Sloth Lauszus. All rights reserved.

 This software may be distributed and modified under the terms of the GNU
 General Public License version 2 (GPL2) as published by the Free Software
 Foundation and appearing in the file GPL2.TXT included in the packaging of
 this file. Please note that GPL2 Section 2[b] requires that all works based
 on this software must also be made publicly available under the terms of
 the GPL2 ("Copyleft").

 Contact information
 -------------------

 Kristian Sloth Lauszus
 Web      :  http://www.lauszus.com
 e-mail   :  lauszus@gmail.com
*/

// Inspired by: https://github.com/arduino/Arduino/blob/master/hardware/
    arduino/avr/cores/arduino/HardwareSerial.cpp and https://github.com/
    arduino/Arduino/blob/master/hardware/arduino/sam/cores/arduino/
    RingBuffer.cpp

#include <stdint.h>
#include <string.h>

#include "ringbuffer.h"

RingBuffer::RingBuffer(void) {
    memset(buffer, 0, BUFFER_SIZE);
    head = 0;
    tail = 0;
}
```

```
30
31 void RingBuffer::write(int c) {
32     int i = (uint32_t)(head + 1) % BUFFER_SIZE ;
33
34     // If we should be storing the received character into the location
35     // just before the tail (meaning that the head would advance to the
36     // current location of the tail), we're about to overflow the buffer
37     // and so we don't write the character or advance the head.
38     if (i != tail) {
39         buffer[head] = c;
40         head = i;
41     }
42 }
43
44 void RingBuffer::clear(void) {
45     head = tail;
46 }
47
48 int RingBuffer::available(void) {
49     return ((uint32_t)(BUFFER_SIZE + head - tail)) % BUFFER_SIZE;
50 }
51
52 int RingBuffer::read(void) {
53     if (head == tail) // If the head isn't ahead of the tail, we don't have
          any characters
54         return -1;
55     else {
56         int c = buffer[tail];
57         tail = (uint32_t)(tail + 1) % BUFFER_SIZE;
58         return c;
59     }
60 }
```

```
1  /* Copyright (C) 2015 Kristian Sloth Lauszus. All rights reserved.
2
3   This software may be distributed and modified under the terms of the GNU
4   General Public License version 2 (GPL2) as published by the Free Software
5   Foundation and appearing in the file GPL2.TXT included in the packaging of
6   this file. Please note that GPL2 Section 2[b] requires that all works based
7   on this software must also be made publicly available under the terms of
8   the GPL2 ("Copyleft").
9
10  Contact information
11  -------------------
12
13  Kristian Sloth Lauszus
14  Web      :  http://www.lauszus.com
15  e-mail   :  lauszus@gmail.com
16 */
17
18 #ifndef __ringbuffer_h__
19 #define __ringbuffer_h__
20
21 #define BUFFER_SIZE 128
22
23 class RingBuffer {
24 public:
25     RingBuffer(void) ;
26     void write(int c);
27     void clear(void);
28     int available(void);
29     int read(void);
30
```

```
31 private:
32     int buffer[BUFFER_SIZE];
33     int head;
34     int tail;
35 };
36
37 #endif
```

Appendix I

Misc header

```
/* Copyright (C) 2015 Kristian Sloth Lauszus. All rights reserved.

 This software may be distributed and modified under the terms of the GNU
 General Public License version 2 (GPL2) as published by the Free Software
 Foundation and appearing in the file GPL2.TXT included in the packaging of
 this file. Please note that GPL2 Section 2[b] requires that all works based
 on this software must also be made publicly available under the terms of
 the GPL2 ("Copyleft").

 Contact information
 -------------------

 Kristian Sloth Lauszus
 Web      :  http://www.lauszus.com
 e-mail   :  lauszus@gmail.com
*/

#ifndef __misc_h__
#define __misc_h__

#define constrain(amt,low,high) ((amt)<(low)?(low):((amt)>(high)?(high):(amt)
    ))

enum Connected {
    CONNECTED_4 = 0,
    CONNECTED_6,
    CONNECTED_8,
};

#endif
```

Appendix J

Makefile

```
CC=g++
CFLAGS+=`pkg-config --cflags opencv`
LDFLAGS+=`pkg-config --libs opencv`

# Link UV4L and WiringPi libraries on ARM
ifneq ($(filter arm%,$(shell uname -m)),)
    CFLAGS+=-I/usr/local/include
    LDFLAGS+=-L/usr/lib/uv4l/uv4lext/armv6l -luv4lext -Wl,-rpath,'/usr/lib/
        uv4l/uv4lext/armv6l' -L/usr/local/lib -lwiringPi
endif

# Change this to the name of your main file
PROG=main

# Create a list of all object files
OBJS = $(addsuffix .o,$(basename $(notdir $(wildcard *.cpp))))

.PHONY: all clean run

$(PROG): $(OBJS)
    mkdir -p bin
    $(CC) -o bin/$(PROG) $(OBJS) $(LDFLAGS)

%.o: %.cpp
    $(CC) -O3 -c $(CFLAGS) $<

all: $(PROG)

# Remove all objects and bin directory
clean:
    rm -rf $(OBJS) bin

```

```
32 # Run executable inside bin directory
33 run: all
34     sudo ./bin/$(PROG)
35 
36 # Rebuild everything when makefile changes.
37 $(OBJS): Makefile
```