Junior Journal

Journal Name

OPEN ACCESS
ARTICLE

Opening OpenCV

Regular Paper

Markus Hovorka[1] , Clemens Jung[1] , Thomas Langenau[1] , Philipp Lütge[2,*], Patrick Podest[1] , Veronika Schrenk[1] and Bruno Tiefengraber[1]

[1] HTBLuVA Wr. Neustadt, Austria
[1] HTBLuVA Wr. Neustadt, Philipp Development Austria
* Corresponding author E-mail: philipp@philipp-development.at

Received D M 2014; Accepted D M 2014

DOI: 10.5772/chapter.doi

Abstract

The ongoing development on image sensors and lenses led to an immense price drop for cameras. The current camera included in the Botball Kit 2014 has a resolution of 640x480 pixel and is available for about 3 Dollars in Chinese online shops[1]. The high availability of hardware creates the need for software which is capable of processing the recorded images and is easy to use. To make image processing feasible for everyone OpenCV was developed. OpenCV is also the heart of the blob tracking library inside of the Botball firmware. This library only supports blob tracking and is therefore very dependent on the current lighting conditions. OpenCV offers a lot of alternatives to blob tracking. This allows us to choose our object detection algorithm depending on the current situation. The following paper will show how to use OpenCV and adapt it to the needs of a Botball-Team.

Keywords

1. Introduction

All Botballkits are equipped with an RGB - camera which is accessible through the official Botball firmware. After a little search we found that the Botball library uses OpenCV to process the recorded images. Being able to use the complete OpenCV library would give an impressive advantage compared to the standard library.

2. Open Source Computer Vision Library[2]

OpenCV is an open source library for computer vision and machine learning. As it is licensed under a BSD license it is easy for businesses to utilize and modify the code[3]. It was built to provide a common infrastructure for computer vision applications and to reduce computing and development time. OpenCV includes more than 2500 optimized algorithms to recognize faces, detect objects, classify actions in videos, track camera movements, track moving objects, extract 3D models from objects, produce 3D point clouds and many more.

2.1. Platforms

OpenCV has native interfaces for:

- C++
- C
- Python
- Java
- MATLAB

on Windows, Linux, Android, iOS and MacOS. There are wrappers for other languages built by the community but they are not officially supported. OpenCV aims for real-time vision application and has therefore support for both CUDA and OpenCL[4]. Using the CUDA implementation can boost the applications performance by up to 30 times for primitive image processing and up to seven times for stereo vision.

3. The current implementation

In the version for the 2014 Botballcompetition the camera-library enables a mechanism called Blob tracking to identify objects.

3.1. Blob tracking

The Blob tracking as it is implemented in the current KIPR Link firmware searches images for regions with one of the configurable colors. The found regions are referred to as Blobs. The library builds a bounding-box around those blobs and makes the following functions available:

- int get_channel_count(void)
- int get_object_count(int channel)
- char* get_object_data(int channel, int object)
- int get_object_data_length(int channel, int object)
- double get_object_confidence(int channel, int object)
- int get_object_area(int channel, int object)
- rectangle get_object_bbox(int channel, int object)
- point2 get_object_centroid(int channel, int object)
- point2 get_object_center(int channel, int object)

3.1.1. Object confidence

As the recognition is highly dependent on the lightning conditions an object usually can not be distinctively classified. Hence, the library use a confidence level between 0 and 1 to describe how significant the object is in the channel. In Fig. 1 the object confidence expresses the green blobs area in relation to its bounding box.

3.1.2. Object center vs. Object centroid

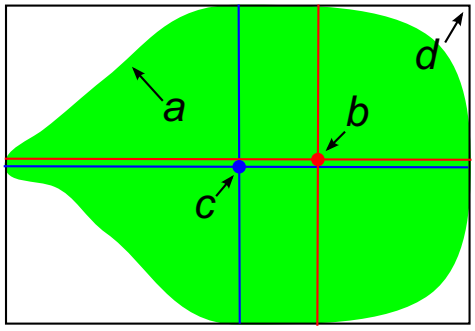


Figure 1. A schematic view of an image processed by the libkovan.

Fig. 1 shows the difference between the center of an object *c* and the centroid *b*. The center is the geometric center of the bounding box *d* where the centroid is the center of the blob *a*. The centroid should be preferred over the center because it is more accurate in terms of finding the blobs position.

3.2. Hue Saturation Value model[5]

Unlike the well known RGB model the HSV model uses only one numeric value to describe a color[6].

Hue

Hue is the representation of the pure color around the color-wheel as seen in Fig.2. Usually hue is expressed as the degree on the wheel starting with red at 0 and moving clockwise with yellow at 60 degrees, green at 120 degrees, cyan at 180 degrees, blue at 240 degrees and magenta at 300 degrees.

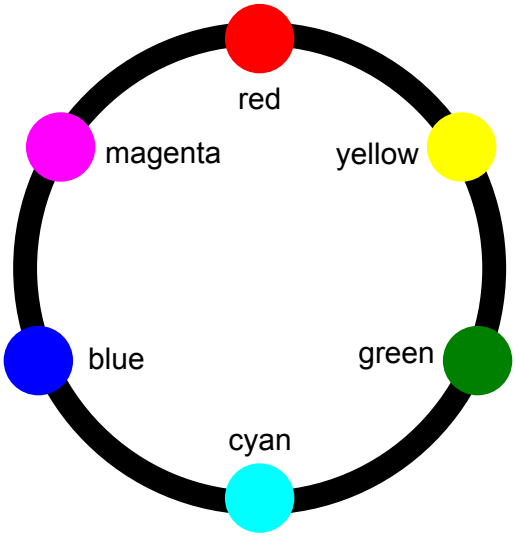


Figure 2. Hue is representation of colors around a color-wheel.

Saturation

As depicted in Fig. 3 the saturation describes how white the color is. Pure red has a saturation of 1, tints of red have saturations less than 1 and white has a saturation of 0.

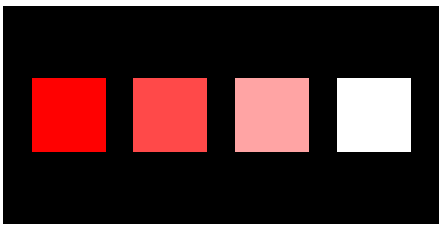


Figure 3. The hue is 0 for all four squares where the saturation is 1, 0.75, 0.25, 0 from left to right.

Value

Value is also called lightness. This value describes how dark a color is with 0 for black and increasing values the lighter a color becomes.

3.3. Problems of the Blob tracking library

Even though the Blob tracking does a fairly good job it is not perfect. The biggest problems are as follows.

Very dependent on lighting conditions

The Blob tracking library applies the saturation and value values in addition to the hue color. This is intended to raise the precision of the object recognition but it makes it dependent to the lighting conditions.

The bounding box

The bounding box makes it easy to access an objects location and size but it gets unprecise when the objects shape is different from a rectangle.

Ignoring shapes

Certain scoring items do have the same color but have a different form. It is hardly possible to distinct a circle from a rectangle using the botball library.

3.4. Improving Blob tracking

The problems mentioned in 3.3 can be solved using only software which is already installed on the KIPR Link. Some of this can even be done using the Blob tracking library.

Light independent color recognition

This is technically impossible but the Blob tracking library can be improved by creating a custom library which simply ignores the saturation and value. It only searches for certain colors and not tints of them. An implementation is depicted in listing 1. The result of this code is shown in Fig. **??**.

Listing 1. This code captures a frame from the default camera and detects red shapes.

```
int main( int argc, char** argv )
{
 // camera-device
 VideoCapture cap(0);
 Mat img_scene;
 Mat hsv;
 Mat processed;
 // records a frame from the camera
 cap >> img_scene;
 // converts the frame from bgr to hsv
 cvtColor(img_scene, hsv, CV_BGR2HSV);

 inRange(hsv, Scalar(0,160,60),
  Scalar(20,256,256), processed);

 imwrite("Thresh.png", processed);
 imwrite("color.png", img_scene);
}
```

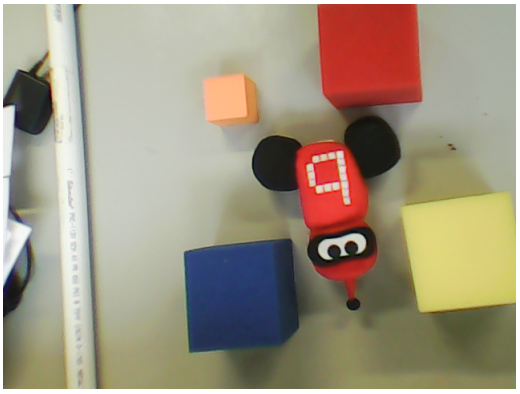
Recognizing an objects shape

There are different approaches which require different software and programming skill levels. Two of them are:

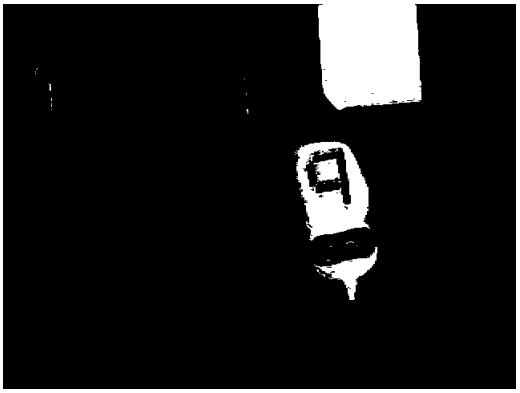
Determining an objects shape using the Blob tracking library

The majority of scoring items are rectangles or circles. Fig. 5 shows that a circle and a rectangle can have the same center, centroid and bounding box even though they are completely different objects. The only difference is the confidence level which is around 100% for the rectangle and is about 85% for the circle. These 85% are calculated as follows:

$$A_{BoundingBox} = (2 * r)^2 \tag{1}$$



(a) The image shows typical Botball scoringitems recorded with a KIPR Link and the provided camera.



(b) This image is processed using the source code from 1

Figure 4. A comparison of the original image *a* and the processed image *b*

$$A_{Circle} = r^2 * \pi \tag{2}$$

$$C = (A_{BoundingBox} - A_{Circle}) * 100 \tag{3}$$
$$C = (((2 * r)^2) - (r^2 * \pi)) * 100 \tag{4}$$
$$C = (4 - \pi) * r^2 * 100 \tag{5}$$
$$C = 85.84 \tag{6}$$

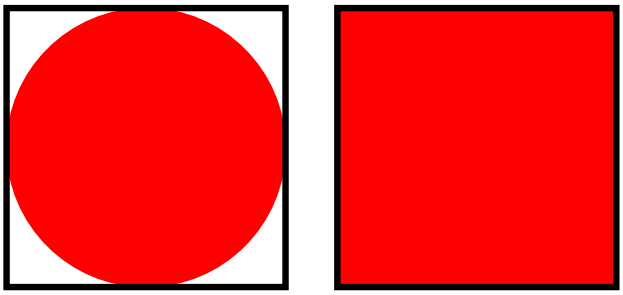


Figure 5. Both bounding boxes are the same size but the circle does not fill the entire bounding box. This difference can be detected using the confidence property.

Determining an objects shape using OpenCV

The following list shows some possibilities to detect objects using OpenCV:

- Feature-matching
- Haar-training
- Line transformation
- Cirlce transformation
- Edge - Detection

All of the above operate on grayscale images.

Edge - Detection

Edge - Detection makes it easy to recognize an objects shape. It also increases an applications performance because the images get converted into binary images[1].

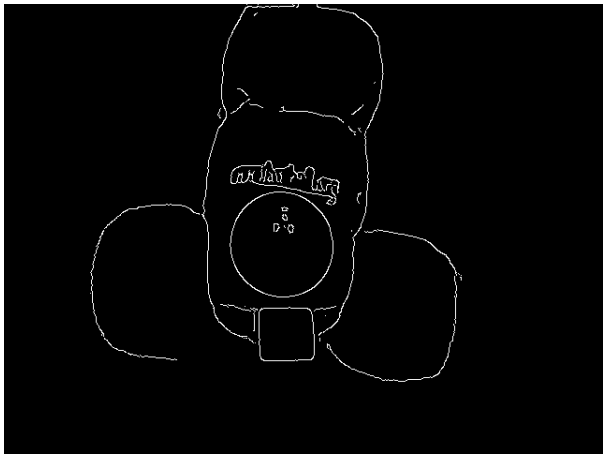


Figure 6. Canny Edge - Detection applied to an image of the botguy.

The Canny Edge Detector[7]

The Canny Edge Detector was developed by John F. Canny in 1986 and is also known as the optimal detector. The Canny algorithm aims to satisfy three main criteria:

- **Low error rate:** Meaning a good detection of only existent edges.
- **Good localization:** The distance between edge pixels detected and real edge pixels have to be minimized.
- **Minimal response:** Only one detector response per edge.

Workflow of an edge - detection application

The first step is converting the color image into a grayscale image. This helps to reduce the needed computing time.

The second step is to filter noises by applying a blur[2]. After that the Canny function from the OpenCV library can be applied. An implementation of this workflow is shown in listing 2.

Listing 2. This code captures a frame from the default camera, blurs it and applies a Canny edge detection.

```
int main( int argc, char** argv )
{
 // camera-device
 VideoCapture cap(0);
 Mat img_scene;
 Mat gray;
 Mat edges;
 // records a frame from the camera
 cap >> img_scene;
 // converts the frame from bgr to grayscale
 cvtColor(img_scene, gray, CV_BGR2GRAY);
 //blur the image
 blur( src_gray, detected_edges, Size(3,3) );
 //apply the edge-detection
 Canny( gray, edges, 100, 300, 3 );

 imwrite("Edges.png", edges);
}
```

Hough Circle Transform[8]

To define a cirle three parameters are needed.

- X center
- Y center
- radius

For sake of efficiency, OpenCV implements a detection method slightly trickier than the standard Hough Transform: *The Hough gradient* method.

To make the detection more stable the image should already be thresholded using an edge-detector.

Listing 3. This code captures a frame from the default camera, blurs it and applies a Canny edge detection. The tresholded image is then applied a circle-transform.

```
int main( int argc, char** argv )
{
 // camera-device
 VideoCapture cap(0);
 Mat img_scene;
 Mat gray;
 Mat edges;
 vector<Vec3f> circles;
 // records a frame from the camera
 cap >> img_scene;
 // converts the frame from bgr to grayscale
 cvtColor(img_scene, gray, CV_BGR2GRAY);
 // blur the image with a kernel of 3
 blur( src_gray, detected_edges, Size(3,3) );
 // apply the edge-detection
 Canny( gray, edges, 100, 300, 3 );

 // Apply the Hough Transform to find the circles
 HoughCircles( edges, circles, CV_HOUGH_GRADIENT,
  1, edges.rows/8, 200, 100, 0, 0 );

 // Draw the circles detected
 for( size_t i = 0; i < circles.size(); i++ )
 {
  Point center(cvRound(circles[i][0]),
         cvRound(circles[i][1]));
  int radius = cvRound(circles[i][2]);
  // circle center
  circle( img_scene, center, 3, Scalar(0,255,0),
         -1, 8, 0 );
  // circle outline
  circle( img_scene, center, radius, Scalar(0,0,255),
         3, 8, 0 );
 }
 imwrite("Circles.png", img_scene);
}
```

[1] A binary image contains only two colors - usually black and white.
[2] usually a Gaussian Blur

The code in listing 3 demonstrates how to detect circles in an image. The detected circles are then overlayed on the original image. Fig. 7 shows an image as it is generated by the code listing.

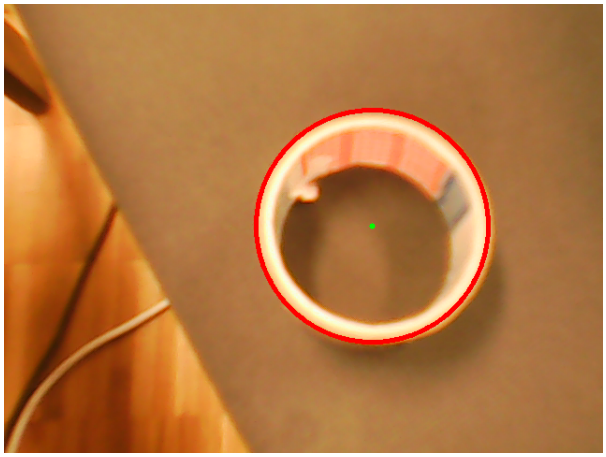


Figure 7. The detected circle is drawn onto the recorded image.

4. Alternatives to Blob tracking

Finding an object in an image is like telling a human what he has to look for. This is usually done with descriptions e.g. "The Botguy has a rectangular body and two round wheels on the bottom." Unfortunately it is very hard for robots to understand a natural language[3]. Hence the objects have to be described in a different way.

4.1. Feature-matching

A common way of describing an object for image analysis is feature description.

4.1.1. Features

At the moment there is no exact definition of what a feature is. A feature can be treated as an "interesting" part of an image. The following types of image features can be distinguished:

- Edges
- Corners
- Blobs
- Ridges

4.1.2. Feature Detector

The most important part of feature matching is the feature detector because it is the base for any further computation. Therefore a stable, fast and reliable algorithm is essential. A very common detector is the SIFT[4] detector. Even though it is not free the better choice is the SURF detector[9].

[3] e.g. english
[4] Scale-invariant feature transform

Speeded Up Robust Feature

Compared to SIFT, SURF is several times faster and is claimed to be more robust against image transformation. An application of the algorithm is patented in the U.S.

4.2. Cascade Classifiers

Cascading is a multiple stage identifier. This increases the performance which makes it possible to use this mechanism in real time applications. Cascading is also reliable with a very low false detection rate. The first use for Cascade Classifiers was introduced in 2001 by Paul Viola and Michael Jones to implement real time face-tracking on low power devices[10].

The basic algorithm

The algorithm for all rectangles in the images can be summed up as follows:

- Stage 1: According to classifier 1 is there an object in the rectangle? If yes: continue; If no: there is no object
- Stage 2: According to classifier 2 is there an object in the rectangle? If yes: continue; If no: there is no object
- ...
- Stage n: According to classifier n is there an object in the rectangle? If yes: continue; If no: there is no object

The question if there is an object is answered by weak learners. None of them can classify an object by itself but they are expressive enough when they are combined to classify any object.

Stage properties

To get a good overall performance each stage must validate all objects. For example: If stage 1 has a false negative rate of 20% the overall detection rate can not be higher than 80% whatever classifiers are applied afterwards.

This concludes that a good classifier should accept all positives but does not have to reject all negatives as there follow other classifers which eliminate the negatives. Speed is also more important than a good negative detection rate as it makes it possible to apply more classifiers in the same time.

The first detector of Viola & Jones had 38 stages, with 1 feature in the first stage, then 10, 25, 25, 50 in the next five stages, for a total of 6000 features. The first stages remove unwanted rectangles rapidly to avoid paying the computational costs of the next stages, so that computational time is spent analyzing deeply the part of the image that have a high probability of containing the object.

Feature types and evaluation

The features employed by the detection framework universally involve the sums of image pixels within rectangular areas. Because they rely on more than one rectangle they are more complex than features used within

the Haar basis function. A features value is always calculated as the sum of pixels in the clear area subtracted from the sum of pixels in the shaded area.

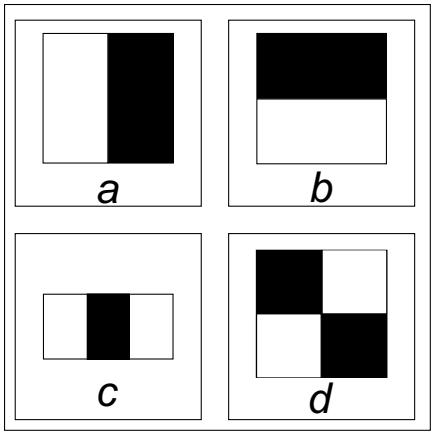


Figure 8. The four feature types as they were defined by Viola and Jones.

Rectangular features are primitive compared to other alternatives like steerable filters[11]. Their result is coarser although they consider horizontal and vertical features. The main benefit of rectangular features is their constant time in evaluation in integral images compared to sophisticated alternatives. The four feature types as they were defined by Viola and Jones are depicted in Fig. 8.

Integral images

Integral images[5] are used for fast calculation of sums of pixels inside rectangular areas.

4.2.1. Cascade classifiers in OpenCV

Although cascade classifiers are complex in theory it is easy to use them as it is only one function call inside OpenCV. The *detectMultiScale* function does all the work of detecting the objects by applying the features defined in a XML-file. OpenCV does already have a configuration file for face detection and comes with all the functions needed to create custom configurations.

4.2.2. Creating your own cascade classifiers

This process is also called Haar-training and it needs two kinds of images to work:

- positive samples: These images contain the object
- negative samples: These images do not contain the object

The first step is to record those images. It is a good practice to organize the images in two folders (positives, negatives). After that a list of files should be created. The list of positive images is slightly different from the list of negatives as it does not only contain the file names but also the amount of objects in the image and their location. To achive good results you should use between 1000 and 3000 positive samples and about twice as many negative samples. The more samples you use the more precise the classifier will be but it causes an significant increase of time for the training process.

Listing 4. This is an example of the file contents for positives and negatives.

```
negatives:
images/negatives/img1.jpg
images/negatives/img2.jpg
images/negatives/img3.jpg
...

positives:
images/positives/img1.jpg 1 0 0 240 320
images/positives/img2.jpg 1 0 0 240 320
images/positives/img3.jpg 1 0 0 240 320
...
```

As depicted in listing 4 the list of positives is structured as follows: image-path amount-of-objects object1-x object1-y object1-width object1-height object2-x ...

The next step is to create samples out of the given images. This should be done using the *opencv_createsamples* programm which is included in the OpenCV installation[12]. This program transforms and rotates the input images to increase the amount of training images and therefore improve the classifiers precision. This program produces a *.vec* file which contains all the information for the last step.

The last step is the *opencv_haartraining* program. This produces the xml-file used by the *detectMultiScale* function. This process can take several hours to finish.

5. 3 Dimensional Botball

The recently added 3D-sensor can help to build incredible robots. As the recorded depth-images of the 3D-sensor can be treated as grayscale images they are easy to process. All of the tools needed to detect three dimensional shapes are already installed on the KIPR Link. All of the possibilities of Chapter 4 Alternatives to Blob tracking can be used on the 3D images as well. The Botball vision library still has enough space for improvements but it is really sophisticated compared to the 3D vision library.

6. Conclusion

A custom vision library has a lot of features which are not present in the standard library. Most of them are complex and require certain knowledge of machine learning. The basic features can be customized to be more stable in special cases.

At the moment the KIPR Link ships with some handy tools which can be powerful in combination with a camera. But the highest level of object recognition can be achieved if the 2D color image is combined with a 3D image. This is not only fun to play with but also the next step into the future of robots.

7. References

[1] The botball camera. http://www.buyincoins.com/item/1891.html, March 2014.

[5] a.k.a Summed area table

[2] Opencv homepage. http://opencv.org, February 2014.
[3] Bsd license. http://www.linfo.org/bsdlicense.html, February 2014.
[4] Cuda support. http://opencv.org/platforms/cuda.html, February 2014.
[5] The hsv model. http://smoumie.blogspot.co.at/2013/12/hsv-preferred-color-model-of-computer.html, March 2014.
[6] The rgb model. https://www.princeton.edu/ achaney/tmve/wiki100k/docs/RGB$_c olor_m odel.html$, *March*2014.
[7] The canny edge detector explained. http://docs.opencv.org/doc/tutorials/imgproc/imgtrans/canny$_d etector/canny_d etector.html$, *March*2014.
[8] http://docs.opencv.org/doc/tutorials/imgproc/imgtrans/hough$_c ircle/hough_c ircle.html$, *March*2014.
[9] Tinne Tuyellars Herbert Bay and Lue Van Gool. Surf: Speeded up robust features.
[10] Paul Viola and Michael Jones. Robust real-time object detection. In *International Journal of Computer Vision*, 2001.
[11] W. T. Freeman and E. H. Adelson. The design and use of steerable filters. *IEEE Transactions on Pattern analysis and machine intelligence*, 13:891Ű906, 1991.
[12] Haar training tutorial. http://note.sonots.com/SciSoftware/haartraining.html, March 2014.