Robotic Object Tracking for Path Planning Using K-means and OpenCV

Dept. of Electrical and Computer Engineering
Northeastern University
Boston, MA 02115

Dept. of Electrical and Computer Engineering
Dept. of Physics
Northeastern University
Boston, MA 02115

Matthew (Sheng-Yang) Goh
Dept. of Electrical and Computer Engineering
Dept. of Bioengineering
Northeastern University
Boston, MA 02115

Dept. of Electrical and Computer Engineering
Northeastern University
Boston, MA 02115

Abstract

Object detection and tracking has a variety of applications, including security and surveillance, human-computer interaction, biometrics, and vehicle navigation [12]. In the specific case of autonomous vehicle navigation, object detection capabilities remain a core problem [5]. Object detection algorithms need to run in near real time so that evasive actions can be successfully performed. At the same time, the implementation of any such solutions are often constrained by external factors, such as economic cost or consumer affordability [5]. The goal for this project was to examine the efficacy of object detection algorithms implemented in OpenCV, and to build an autonomous robotic system. We focused on the ability of k-means, Canny edge detection and OpenCV's (Open Source Computer Vision) *findContours* algorithm to navigate an iRobot Braava towards a goal object while also navigating around avoidance obstacles [11].

1 Introduction

In recent years, large efforts have been spent on developing self-driving car systems to reduce traffic congestion and improve traffic safety. Autonomous vehicles require accurate up to date information about their surroundings in order to safely navigate the environment. They must detect and perceive objects around them from trained data to be able to distinguish between obstacles that should be avoided and terrain that should be driven. Self driving vehicles, including one in development by Google, rely on a Light Detection and Ranging (LIDAR) system, GPS and cameras to detect and verify the classification of objects around them [4]

Our objective was to build an autonomous robotic car that can execute a set of maneuvers based on a classification algorithm programmed into an on-board computer and input from a small mounted camera. While self-driving car systems are much more complex in reality, our goal was to build a simplified model to explore the efficacy of various algorithms. In particular, we focused on k-means, Canny edge detection, and OpenCV's (Open Source Computer Vision) *findContours* algorithm [11]. In our implementation, Canny Edge detection and *findContours* are used in image segmentation while k-means is used for object classification.

1.1 K-means

K-means is a widely used unsupervised clustering and classification algorithm. It groups data points into *k* clusters based on the closest cluster mean (centroid) to each point. It is comprised of three steps: initialization, classification, and centroid calculation, which are repeated until convergence is reached. In the initialization step, the data set, number of clusters (*k*) and starting means are each defined. The initial values for the mean of each class are typically determined by selecting *k* random values from the data set. In the assignment step, each data point is assigned a class label based on euclidean distance to the nearest cluster centroid. The arithmetic mean for each cluster is then recomputed as the averaged values of all data points belonging to that cluster. This process repeats until the convergence condition is reached [3].

1.2 OpenCV

For the object classification tasks, we utilized the OpenCV library of programming functions developed for real-time computer vision applications. These include but are not limited to human computer interaction, object recognition, face recognition, and motion tracking. C, C++, Java, and Python languages are among several supported implementations. Due to its flexibility and widespread use, it was a suitable library for our project.

1.3 Canny Edge

To classify an object, the object must first be isolated from the rest of the image. The goal of edge detection is to locate sharp changes in image brightness or points at which the image contains discontinuities [8]. Typical edge detection algorithms are based on the first and second order derivative method of image intensity and include filtering, enhancement, and detection. Canny edge detection is one such segmentation algorithm that utilizes hysteresis thresholding.

The algorithm of Canny edge detection was designed by John F. Canny in 1986 with the following criteria in mind [6, 9]:

1. Low error rate: The ability to identify as many of the true edges as possible while at the same time reducing noise produced by false positives.
2. Good Localization: Decrease the difference between the detected and true edge locations.
3. Minimum response: Identify edges one by one and ensure that any noise will not indicate a false positive.

Canny edge detection involves four main steps: intensity gradient calculation, non-maximum suppression and hysteresis thresholding [9].

1. **Noise Reduction**: Noise can negatively impact the performance of edge detection since the algorithm is primarily based on first and second order derivatives. Thus, the first step is to reduce and remove the noise in the image by applying a 5×5 Gaussian filter.
2. **Intensity Gradient Calculation**: A Sobel kernel is then used to filter the smoothed image to calculate the first derivative in the horizontal and vertical directions. The derivatives are denoted as G_x and G_y respectively. The edge gradient and direction for each pixel are calculated utilizing these two images and the following equations where G (eq. 1) is equal to the edge gradient and θ (eq. 2) is equal to the angle.

$$G = \sqrt{G_x^2 + G_y^2} \tag{1}$$

$$\theta = \tan^{-1}(\frac{G_x}{G_y}) \tag{2}$$

   The direction of the gradient is always perpendicular to the edges and rounded to one of four angles representing vertical, horizontal or one of two diagonal directions.
3. **Non-maximum Suppression**: Any unwanted pixels that may not make up an edge are removed from the image. This is determined by checking if each pixel is a local maximum of its neighborhood in the gradient direction. Any pixels that are not identified as a local maximum value are set to zero while pixels that are determined to be local maxima are used as input to the next step of the algorithm.

4. **Hysteresis Thresholding**: In this step, two threshold values, minVal and maxVal, are utilized to complete edge detection for the image. All edge pixels with an intensity gradient value that is greater than maxVal are immediately accepted as edges while those below minVal are rejected. Edges with intensity gradient value between the two threshold values are classified as edges if they connect to definite edge pixels, and are rejected otherwise. This enables detection of edge portions that are not as distinct in the image as others.

Figure 1 shows the progression of the first two steps of the Canny edge detection algorithm. On the left, a Gaussian filter is applied to an image of a Cheez-It box. The center and right most images display the horizontal and vertical derivatives of the image respectively.

Figure 1: The first steps of the Canny Edge algorithm in OpenCV

1.4 findContours

The OpenCV *findContours* function was designed to detect contours in a binary image as well as its hierarchy. Contours are characterized as a curve that connects continuous points constituting the boundary of an object. The function inputs are the image of interest processed with Canny, followed by two parameters specifying the contour retrieval mode and contour approximation method [9].

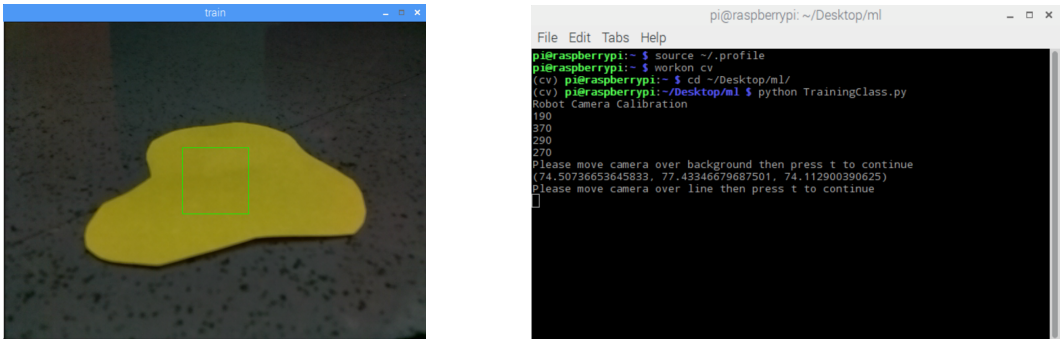
The *findContours* algorithm follows the methods of border following with topographical analysis abilities as developed by Suzuki and Abe in 1983 [9, 11]. The algorithm begins by scanning the image until a border pixel is detected. The parent of the current border is then determined by the type of the current border. (i.e. whether it is an outer or hole border) The border is then followed by inspecting the value (1 or 0) of each pixel surrounding the current border pixel. Once the border traversal returns to the starting point of the border or the pixel of interest is surrounded by solely 0-pixels, the algorithm returns to the point one pixel to the right of which the border was first detected. This algorithm continues until the pixel at the lower right hand corner of the image is reached [11]. The output of the algorithm is an image containing ones where there is a border and zeros elsewhere.

2 Methods

The OpenCV (Open Source Computer Vision) library was installed on a Raspberry Pi 3 computer. The computer was interfaced with an Arduino microcontroller that was programmed to output motor and navigation instructions to an iRobot Braava serving as the robot chassis. With video capture from a Pi camera module, we used an object's color as the criteria for goal or avoidance-directed movement. The movements were also confined to an area marked by boundary lines. In summary, our classification program was designed to perform the following:

- recognize a potential region of interest (ROI) based on contours extracted from Canny edge filtered images
- determine if the ROI reached the minimum size threshold based on its area
- classify `background`, `lines`, `obstacles`, and `goal` objects based on RGB values using k-means
- output motor control instructions to robot based on determined object class

The Arduino microcontroller was programmed to perform the following movements:



(a) Camera output during training

(b) Terminal prompts during training

Figure 2: During training the terminal prompts you for object type, here shown asking for line. Then the user must put the line color (yellow) in the drawn box on the screen and press "t."

- 180 degree turn at **yellow** `lines`
- move around **red** `avoidance` obstacles
- move forward when no objects are detected (`background`)
- stop when the **white** `goal` is reached

2.1 Training

To perform the object detection and classification tasks, we constructed a training program that implemented k-means to learn the colors of objects that were positioned within the bounding box of the video capture frame. The feature vector for k-means comprised of the average RGB value of each object class (`avoid`, `line`, or `goal`).

A bounding box of size 200×200 pixels was created and displayed in the video capture frame. During training, the user is prompted to position the object of interest so that it occupies the entire bounding box. The user is then prompted to press "t" to train the classifier on that object.

For each object classification, the average RGB value of the object, or region of interest (ROI), utilizes the mask for the ROI. This average value is saved as the initial mean for that class. Detected objects are assigned to a class by the minimum euclidean distance calculated between the point of question and each cluster centroid. This implementation allows for flexibility in training. We are able to modify the label of each object that is detected, enabling us to test a variety of objects. We have the ability to show the robot an item and train it to understand the meaning behind that object. For example, when the object is recognized as the trained `avoid` object the robot will respond according to the corresponding actions.

Figure 2 shows the video and command prompt display during the training process. As shown in Figure 2b, the user is prompted to place a line (yellow) object in the green bounding box of the camera. The mean RGB value of the object in the box is then computed and stored as the initial line cluster mean.

2.2 Image Processing and Classification

Our classification program applied Canny edge detection and OpenCV's *findContours* algorithm, which is an implementation of border following algorithms developed by Suzuki and Abe (1985) [11]. With the aid of image filters, we were able to use these methods to extract data from the image and perform k-means classification.

In the initial step, the raw image is converted to gray scale, which optimizes image processing performance by reducing the three color channels (RGB) to a single channel. A bilateral filter is then applied to the image to reduce noise and sharpen edges. This process is similar to Gaussian filtering, which travels across an image to smooth surrounding pixels. One difference, however, is that a bilateral filter also considers intensity of the surrounding pixels, only smoothing the ones of

similar intensity to the central pixel. This maintains sharp drops in contrast (edges) while smoothing similar contrast areas [9].

After the image filtering we then use OpenCV's built in Canny edge detection code. As explained previously, this code first preforms noise reduction with a 5×5 Gaussian filter, followed by an x-axis and y-axis Sobel filter to find the edge gradient and direction of each pixel. It then applies non-maximum suppression in order to remove pixels that do not constitute an edge. The resulting image is a crisp outline of every edge detected.

An example of the gray scale to bilateral filter to Canny edge process can be seen in figure 3. The first image shows the gray scale conversion. The second image (center) is bilaterally filtered, which is noticeably smoother on the inside of the Cheez-it – in the gray scale image, the salt on the cracker is visible, but disappears once smoothing has been applied. Finally, the Canny edge detector identifies the outline of the Cheez-it as shown on the right.

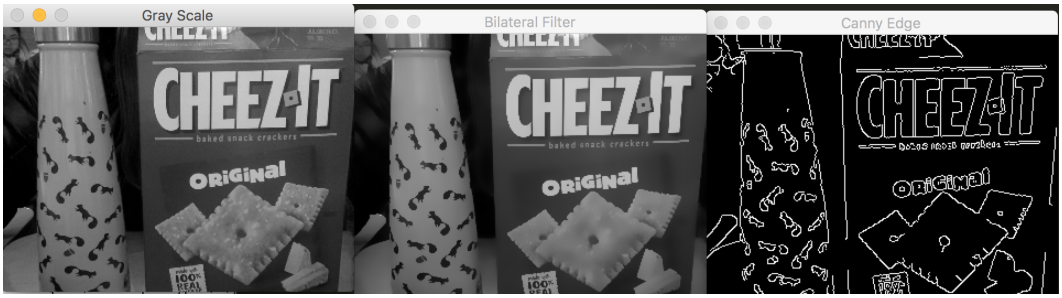
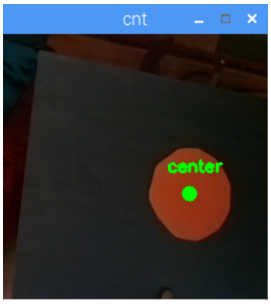


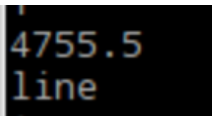
Figure 3: outputs of each step of image filtering

After the Canny edge detection draws the outlines of the image, the OpenCV function *findContours* is used to isolate the objects outlined by Canny. The identified contours are then ordered from largest to smallest based on area. A typical image in our video capture contained around 350 contours, most being very small. Once the contours were ordered based on area, we looked only at the largest contour. Since our camera always faced downwards at a uniform, tiled surface, any object placed on the tiled ground would contribute to the largest contour. In the absence of any objects, the only detectable contours would originate from the tiled ground, which were consistently very small. Thus, we applied a minimum threshold that instructed the algorithm to ignore contours below the threshold. This served as the basis for the "no object" condition, in which our robot would continue to drive straight.

If a contour reached our minimum area threshold, a mask was created that consisted only of the area inside the contour. All of our data was then extracted from this mask. We computed the center of mass of the mask in order to know where in our image the object was located, and we measured the mean pixel RGB inside the mask, which was then sent to the k-means algorithm for classification.



(a) Camera output during classification



(b) Terminal output during classification

Figure 4: We print the center of the object to the screen in order to determine the object location.The area of the object is printed to the terminal, along with the classification of the object

Figure 4 shows the classification code in action. At this point we had `line` trained as orange and not yellow. You can see that the image processing has identified the center of the masked object, and that the k-means algorithm has identified the color as `line`.

3 Results and Challenges

The training phase consisted of capturing an image of the `background` (black), `line` (yellow), `avoid` (red), and `goal` (green) objects that were each positioned within the bounding box of the camera frame. The mean RGB values for the object were computed and served as the initial centroids for k-means clustering. In general, the trained classifier performed with a high degree of accuracy.

However, we found that inconsistent lighting conditions between training and testing environments significantly affected the classifier's performance. Figures 5, 6, and 7 display the variation in learned RGB values across different lighting conditions. In each figure, the left most side of the image displays the learned RGB with minimal lighting. The right most side of the image displays the learned RGB with maximal lighting. Thus, the entire image displays the gradient of all possible learned RGB values between the two conditions. The region bounded within the dotted lines indicate the learned RGB under our most typical lighting condition.

In the the `avoid` object case, test runs in darker environments biased the interpreted RGB value (in either train or test cases) towards black, and so in those cases, an `avoid` object would frequently be misclassified as `background`. In the `line` case, insufficient lighting biased the interpreted RGB value towards a darker yellow that would occasionally be too similar to red-orange, and would cause the classifier to misclassify a `line` object as `avoid`. In situations where lighting was too abundant, the interpreted RGB biased towards white, causing the classifier to interpret a `line` object as `goal`. For the `goal` case, the classification became problematic only in darker environments that caused the RGB to be interpreted as `background`.

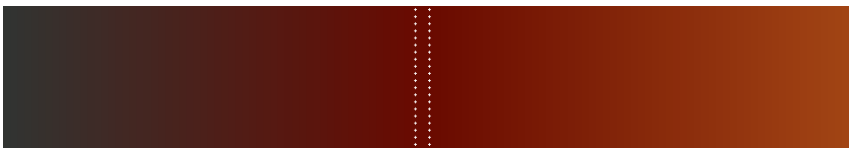


Figure 5: Effect of lighting on the captured RGB of the AVOID object

Figure 6: Effect of lighting on the captured RGB of the LINE object

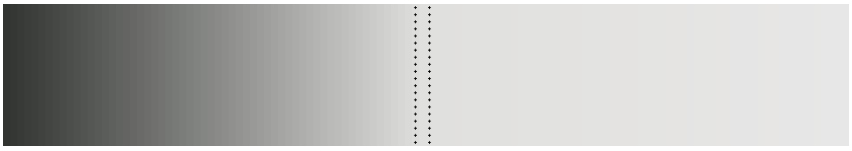


Figure 7: Effect of lighting on the captured RGB of the GOAL object

Given these results, we concluded that RGB values do not provide the best feature set for k-means classification. In the literature, we found that others have also examined and compared the use of RGB color space to HSV (hue, saturation, value) color space in computer vision applications. HSV differs from RGB in that HSV separates out the intensity (luminance) from the color information (chromaticity) [10]. This provides greater robustness to lighting changes and shadows, and also allows for equalization of intensity information without affecting the color component. Had more time been available for this project, one major avenue for improvement would have been examining the performance of HSV compared to RGB.

Another technical challenge that was encountered was the slow framerate of the Pi Camera Module's video capture. The camera module itself is capable of displaying 30fps at 1080p, 60 fps at 720p, and 90fps at 640×480p. However, even at the lowest setting (640×480p), performance of the camera severely suffered when running our classification algorithm. The frames consistently lagged behind real time by 3-4 seconds. Therefore, at the cost of the camera's field of view, we reduced the size of the video capture frame to 240×240 pixels, which alleviated some processing burden on the Pi and reduced the frame lag to roughly 1/4 of a second.

In addition, the robot's velocity set by the Arduino microcontroller needed to be reduced to accommodate for the lag in video capture. During our initial test runs, the robot would reach obstacles before the camera was able to update the frame and detect the object.

In real life applications, this is a critical issue that must be addressed. In the `line` and `avoid` object cases, our robot was programmed to either turn 180 degrees or move around the obstacle, which is analogous to scenarios where a self-driving car must, for example, quickly avoid a pedestrian or physical barrier. One significant difference with our model is that real world self-driving systems contain much greater processing power. For example, Tesla Motors' Autopilot™utilizes the NVIDIA DRIVE PX2 AI computing platform that is optimized for deep learning and autonomous driving [1, 2]. By comparison, the Raspberry Pi's graphics processing chip is a Broadcom VideoCore IV designed for mobile multimedia applications. See Table 1 for general hardware specification comparisons.

Table 1: Comparison of GPU processing power. The Broadcom VideoCore IV is the graphics processing chip in the Raspberry Pi 3. The NVIDIA GTX 1080 is a high-end consumer level desktop graphics card. The NVIDIA DRIVE PX 2 is the graphics computing platform available in multicore configurations used by Tesla Motors. FLOPS = floating point operations per second. TDP = thermal design power

GPU Performance Comparison		
Name	FLOPS	TDP
Broadcom VideoCore IV	24 gigaFLOPS	not listed
NVIDIA GTX 1080	8.9 teraFLOPS	180W
NVIDIA DRIVE PX 2	16 teraFLOPS	250W

4 Future Work

In mobile robotics, Simultaneous Localization and Mapping (SLAM) is a fundamental problem that asks if a robot can explore an unknown environment, reliably build a consistent map of its surroundings, and deduce its location on that map [7]. Algorithms based on both visual and odometry inputs are known as vSLAM (Visual Simultaneous Localization and Mapping) systems. Figure 8 shows a schematic of a vSLAM system (please refer to Karlson et al (2005) for a more detailed overview). The general concept is that vSLAM systems incorporate both visual and odometry sensors to provide dead reckoning data and images of the environment [7]. These data are then combined in the `SLAM` module of Figure 8 to map out landmarks and to maintain estimates of the robot's location within the generated map. Our system contains only visual input but no encoders to provide odometry information, i.e. it is an open loop system with no control feedback. Consequently, our robot's movements were negatively impacted by changes in friction between the wheels and surface. Thus, for future work, a sensible next-step would be to implement encoders to track wheel position and to implement a feedback system for better navigation control.

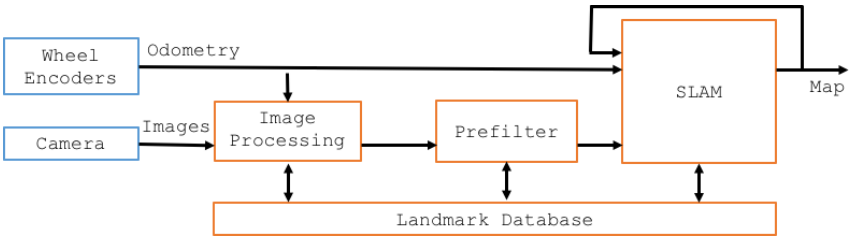


Figure 8: Schematic of a vSLAM system

5 Conclusions

Through this project, we built an autonomous robotic vehicle to safely traverse from a start to goal position using an object detection and classification algorithm. The robotic car was adapted from an iRobot Braava. Controls were implemented using an Arduino for motion and a Raspberry Pi 3 with a camera module for object classification. The objective was to understand machine learning algorithms as applied in computer vision and self driving car design. K-means was utilized in the training and classification steps. Input to the k-means algorithm was the output of image segmentation completed by converting the image to grayscale, applying a bilateral filter, completing Canny edge detection, and utilizing the *findContours* OpenCV function. With the algorithms and implementation described, the robot was able to successfully identify and react to an `avoid`, `line`, and `goal` object.

5.1 Lessons Learned

In order to detect objects, the image first needed to be pre-processed, which involved segmenting the object of interest from the background. We saw that it was necessary to first decrease the noise in images by applying a bilateral filter to increase Canny edge detection accuracy. Without filtering, there would have been a higher probability of detecting false positive edges. The OpenCV *findContours* function is based on border following using topographical edge data in the image. In order to recognize objects of interest, we needed to isolate the largest edges and contours from the entire image.

By utilizing OpenCV on a Raspberry Pi for image processing, we saw the need for optimized algorithms and hardware, as well as learned how to leverage these platforms for our robotic system. In our first tests, the frame rate was not fast enough for our application. However, if the size of the frame was decreased, the frame rate would increase. In this case, less pixels would need processing. In future applications, different chips can be selected for use.

In the object classification step, our approach of using RGB values as the feature vectors for k-means classification was not sufficient to train a reliable classifier. As discussed in the Results and Challenges section, a better approach would be to utilize the HSV color space that is capable of distinguishing chromaticity and luminosity. This would allow us to better account for lighting changes between training and test environments. These challenges also allowed us to better appreciate the complex systems implemented in real-world self-driving systems, which combine cameras, ultrasonics, radar, and LIDAR. All knowledge gained through this project can be brought to the design of future projects, as well as to build upon the abilities of our robotic car.

References

[1] Introducing the new nvidia drive px 2 for autocruise driving and hd mapping. `http://www.nvidia.com/object/drive-px.html`. Accessed: 2016-12-12.

[2] Tesla motors' self-driving car "supercomputer" powered by nvidia drive px 2 technology. `https://blogs.nvidia.com/blog/2016/10/20/tesla-motors-self-driving`. Accessed: 2016-12-12.

[3] D. Arthur and S. Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007.

[4] M. Birdsall. Google and ITE: The Road Ahead for Self-Driving Cars. *Institute of Transportation Engineers*, 84(5):36–39, 2014.

[5] C. Hane, T. Sattler, and M. Pollefeys. Obstacle detection for self-driving cars using only monocular cameras and wheel odometry. *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on, Hamburg, 2015*, 2015. doi: http://dx.doi.org/10.1109/IROS.2015.7354095.

[6] S. R. Joshi and R. Koju. Study and comparison of edge detection algorithms. In *2012 Third Asian Himalayas International Conference on Internet*, pages 1–5. IEEE, 2012.

[7] N. Karlsson, E. di Bernardo, J. Ostrowski, L. Goncalves, P. Pirjanian, and M. E. Munich. The vslam algorithm for robust localization and mapping. In *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, pages 24–29, April 2005. doi: 10.1109/ROBOT.2005.1570091.

[8] R. Medina-Carnicer, R. Munoz-Salinas, E. Yeguas-Bolivar, and L. Diaz-Mas. A novel method to look for the hysteresis thresholds for the canny edge detector. *Pattern Recognition*, 44(6): 1201–1211, 2011.

[9] A. Mordvintsev and K. Abid. Opencv-python tutorials documentation, 2014.

[10] S. Sural, G. Qian, and S. Pramanik. Segmentation and Histogram Generation Using the HSV Color Space for Image Retrieval. *International Conference on Image Processing (ICIP)*, pages 589–592, 2002. doi: http://dx.doi.org/10.1016/0734-189X(85)90016-7.

[11] S. Suzuki and K. Abe. Topological structural analysis of digitized binary images by border following. *Computer Vision, Graphics, and Image Processing*, 30(1), 1985. doi: http://dx.doi.org/10.1016/0734-189X(85)90016-7.

[12] A. Yilmaz, O. Javad, and M. Shaw. Object Tracking: A Survery. *ACM Comput. Surv.*, 38, 2006. doi: http://doi.acm.org/10.1145/1177352.1177355.