Abstract—One of the tasks that image processing tries to solve is to detect objects by looking at the colors of the pixels in an image. A blob detector is a program that connects adjacent similar-colored pixels into regions called blobs. In RGB space, similar-colored pixels form clusters which are found by the program using the K-means clustering algorithm.

I. INTRODUCTION

This year we participated in a robotics competition where image processing was an important part. In the beginning, we used a software library called OpenCV to detect objects in view. We soon discovered that it wasn’t effective enough at recognizing distant balls which was essential in our use case. This motivated us to try different approaches.

During the robotics course, our best approach was using OpenCL[1] to assign pixels into categories based on manually selected color ranges. This greatly improved both the speed of computations and the detection ability of distant balls for our robot. By parallelizing the code using OpenCL we were able to reduce the computation time on a single frame by a factor of three.

For this project we started with our blob detection algorithm from the beginning in order to reduce the amount of manual work needed to select a color configuration and to improve the blob detection. In addition, we created our own thresholding interface.

II. TRESHOLDING

Before starting to use the blob detector, the user must select colors that they wish to be detected. To start thresholding, the user needs a test image where the colors can be extracted from. An example image for our use case is shown on figure 1.

![Fig. 1. Raw image captured by webcam](image)

Secondly, the user must select the number of clusters in the clustering algorithm. This number depends on the task at hand. In a robotics competition, for example, the colors in the environment would be well-defined and the number of clusters would equal the number of different colors on the field.

We implemented the clustering method using OpenCL and K-means algorithm. In the K-means algorithm our categories are colors with centroids in 3-dimensional RGB space. The initial centroids are randomly generated with RGB values ranging from 0 to 255. On each iteration, pixels are assigned to clusters in parallel using OpenCL, after which new centroid values are calculated. After a sufficient amount of iterations, the clusters converge to colors that are most prominent on the source image. The centroids are shown as colored squares in the thresholding interface as shown on figure 2 which the user can click on to toggle whether they wish for the program to detect blobs of that color.

When this process is complete, the thresholding interface displays an image where each pixel’s color is overwritten as the color of its centroid.

![Fig. 2. Thresholding with K-means](image)

At this stage, the user can select colors that the blob detector should detect and save the results in a configuration file. On figure 3, the user has selected three colors and the interface only shows pixels that were assigned to those clusters. This way the user can verify whether the thresholding was successful and retry with different clustering parameters if necessary.

![Fig. 3. Selecting colors](image)
III. Blob detection

In order to make it relatively fast and easy to connect pixels into blobs, all the pixels are first assigned to clusters based on their RGB values. After that is established, each pixel is related to the color of the centroid of this cluster. Now the program is ready to start connecting same-colored pixels into blobs.

The processed image is used to find blobs. The detector reads image data from left to right, starting with the top row. If there is a sequence of pixels that have been assigned to the same color region, they form a chain. All of these chains are potential blobs that could grow on the lower rows.

While processing the rows, the chains that formed on the previous row are kept in memory until after the current row has been processed. This enables to check whether two chains on concurrent rows are touching. If this is the case, then the new chain is appended to the potential blob from the previous row.

Once the whole image has been gone through, the potential blobs that consist of only one chain are discarded and the others are kept in memory as the detected blobs. The user can now do further tests on these blobs to determine their shape on screen and deduct real-world parameters if needed.

On figure 4 it is shown how the blob detector can be used to find balls on a field. Unfortunately, there are also some false blobs detected as seen on the top-right of the figure. In this case, these blobs could be discarded by taking into account the camera angle and the real-world sizes of the balls, but it is a problem that we need to work on.

IV. Discerning similar colors

On figure 5, there is a matchbox with a yellowish central part and a piece of melon that has a similar color. Using the blob detecting algorithm, the pixels in these regions could belong to multiple clusters.

To discern colors that are in multiple ranges, the algorithm selects the range with the centroid closest to the pixel value in RGB space. Figures 6 shows the difference between choosing a random centroid and the closest centroid. It can be seen that the matchbox on the right has a purer color.

V. Conclusion

We were very surprised of how well some parts of our code worked - for instance how well the clustering algorithm selected the colors during the thresholding process. On the other hand, we still have some unresolved problems that we hope to fix during the months ahead perhaps by parallelizing blob connection so we could improve the speed of our algorithm even further.

At this point, the computation time per frame is 33 ms, which is a decent value for a football playing robot.

REFERENCES

https://www.khronos.org/registry/OpenCL/specs/opencl-1.1.pdf