Lab4: Histogram-based single object tracking

Tamás BUKITS and Balázs Márk BÓDIS

I. INTRODUCTION

In this pair programming exercise we implemented a single object tracking algorithm based on color and gradients. Leading up to this task we had two individual parts. One dealt with the candidate selection and matching for the tracker. The other was histogram generation for the description of the candidates. We combined the two individual parts and conducted analysis over real data. We tune the sequences for both color-based and gradient-based tracking.

II. METHOD AND IMPLEMENTATION

The feature selection logic has been developed for being able to expand the code with other features as well. For handling the feature selection a base class, **Feature** was created and more classes (eg.: **Gray**) for each feature which are inherited from it. The functionality of each features (switching to color space or calculating the HOG) are overwritten in the inherited classes functions avoiding using switch cases. In this way the code is clean is suitable for adding extra features as well in the feature. The classes are written in one .hpp file, called **objectTracking**. The main functions and classes are discussed in each section.

A. Color space and channel selection

For each channel classes were created which are inherited form the base, **Feature** class. For the color conversion the OpenCV built in function **cvtColor** was used. This functionality was developed as a virtual function, called generateHistogram which is overwritten in every classes.

Gray color space:

Class name: **Gray**. The gray color space, also known as grayscale or black-and-white, represents images using only shades of gray. In this color space, each pixel is typically represented by a single intensity value, rather than separate red, green, and blue components. The range for intensity values in the gray color space depends on the number of bits used to represent each pixel. The most common range is from 0 to 255, using 8 bits per pixel

HSV color space:

- Class name: **H_color**: The hue component represents the color itself and is typically represented as an angle around a color wheel. Range: [0, 179]
- Class name: **S_color**: The saturation component represents the intensity or purity of the color. It determines how much of the pure hue is present in the color. Range: [0, 255]
- Class name: **V_color**: The value component represents the brightness or lightness of the color. It determines how much light is emitted or reflected by the color. Range: [0, 255]

RGB color space (BGR in OpenCV): The RGB color model represents colors using three components: red, green, and blue. Each component is typically represented by an 8-bit integer value ranging from 0 to 255, where 0 represents the absence of that color and 255 represents the maximum intensity of that color.

- Class name: **R_color**
- Class name: **G_color**
- Class name: **B_color**

B. Histogram Creation and presentation

The logic was implemented in the **initHistogram** function which calculates and visualizes the histogram of a given input frame using OpenCV functions. It defines an array containing the lower and upper range values for the histogram bins. Then it calculates the histogram using the **calcHist** function and normalizes the histogram values using the **cv::sum** function and divides each element of hist by the sum. Finally, it returns the resulting histogram image for drawing as an output.

C. Histogram of Gradients

Class name: **gradient**. Histogram of Gradients (HOG) operates by counting the occurrences of gradient orientations within localized portions of an image. In this specific implementation, the HOG descriptors are computed in the **computeHOGs** function using built-in methods from the **HOGdescriptor** class [2] in OpenCV. An object is created to utilize the functionalities of the HOGdescriptor class. If the size of the patch (region of interest) is different from the desired dimensions of 64 x 128, it is resized accordingly. Subsequently, the resized patch is provided as input to the **compute** function which computes the descriptors of the frame.

D. Candidate Proposal and Decision

We then create the candidate grid and store the locations in a vector. For each candidate we compare the distance between it and the model of the tracked object. When the distance calculation is called, where we have to calculate the differences between the model and the candidates, we put this function into the Feature class, In this way could achieve that that the candidates know which feature was used for calculating the model. We didn't have to write an another switch case for deciding which feature was used as a model.

E. Hyperparameters

In our implementation we have the following five hyper-parameters:

- **number of bins**: the number of bins for the histogram used for the description of the candidates and the model.
- **number of candidates**: the number of candidates proposed for the current frame. The candidates are arranged in a grid around the previous location.
- **grid spacing**: the spacing between the candidates. This is a factor of the candidate size.
- **feature**: the feature used to describe the candidates and the model.
- **starting frame for bb**: the bounding box size is kept constant for the entire sequence. This parameter offers an option to set the bounding box size to any of the groundtruth boundig box sizes from the sequence. This feature was kept as 1 for all testing.

III. DATA

Here we will provide a quick review of the input sequences used for testing the performance of the different approaches.

Task 4.2

”bolt1”: This is a video of Usain Bolt winning a race. The race is a short race and bolt relatively retains the same distance thorough the sequence. However the bbox size changes from the starting size.

Task 4.3

sphere: a ”transparent” orb is handled by a person. There is a blue/green screen effect on the sphere, probably as it acts like the background is visible through the person handling the orb. This causes the transparent effect.

car1: dash-cam footage of a white car on the road. The camera is very unstable throughout the whole video.

Task 4.4

basketball: a game of basketball is being played. The objective is to track Derrick White (in green) as the Boston Celtics are defending against the Toronto Raptors. The camera is fairly stable and has a continuous motion.

ball2: we are tracking a football as it is approaching the goal. The main problem is caused by the texture of the net behind the ball. The camera is static and the ball is already near the goal so the size does not change too much during the sequence.

IV. EVALUATION

Task 4.2

We used the bolt1 sequence for evaluation. The tracking works well for each feature before frame 160 for all of the features but H. After this frame the runners get close to each other and the tracking gets lost with another runner. For the **H** feature we got our best result: 0.529787. Other results can be seen in *Table 1*. We used 16 candidates and 16 bins with each feature. In *Figure 1.* we can see some frames of the sequence, in the first frame displayed we can see an instance

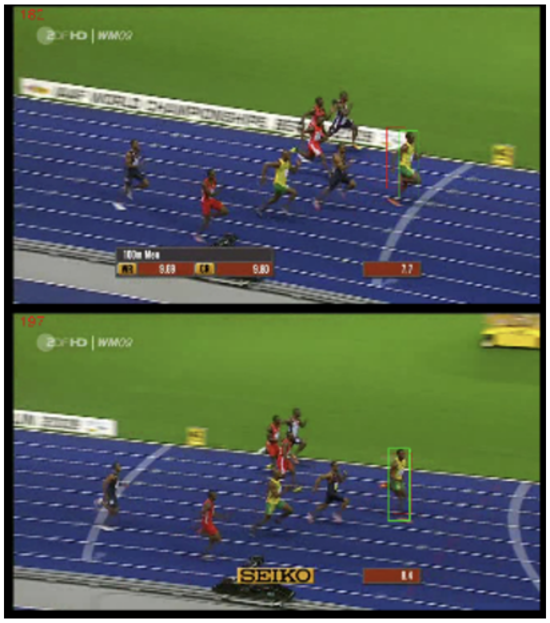

Fig. 1. Color based tracking (grayscale on top, Hue on bottom) for the bolt1 sequence

Feature	Accuracy
Gray	0.14649
R	0.196854
G	0.213257
B	0.17849
H	**0.529787**
S	0.216313
V	0.178547
HOG	0.220012

TABLE I
MEASURED ACCURACY FOR DIFFERENT FEATURES

when grayscale based matching lags behind the subject. In the second displayed frame we can see the H feature based tracking.

Task 4.3

In this task we tested the color-based features for tracking. We implemented the following 6 color based features: Grayscale, Red channel, Green channel, Blue channel, Hue and Saturation.

sphere: we started by testing which feature works best for this sequence. We fixed the number of candidates to 81 and the number of bins to 16. Grid spacing was set to 0.05. We tested the different color based features, the results can be seen in *Table 2.*. Gray-scale performed best and the average time/frame was also very competitive with the faster features (single RGB color channels). We moved forward with optimizing the number of candidates. The time/frame is proportional to the number of candidates so we did not test a very high amount of candidates. The results can be seen in *Table 3.* We concluded that while testing with over 200 candidates could improve performance but we omitted this because of the performance costs. We then tested for the best grid spacing, our results can be seen in *Table 4.*

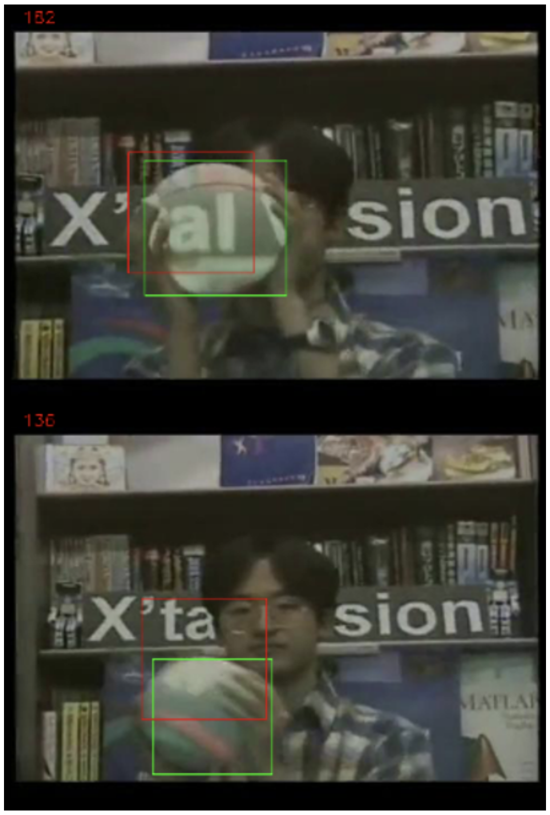

Fig. 2. Example frames from the sphere sequence

Feature	Accuracy	Time
Gray	**0.472579**	**5.57687 ms/frame**
R	0.435545	4.45635 ms/frame
G	0.436058	4.47126 ms/frame
B	0.423031	4.63848 ms/frame
H	0.276991	15.231 ms/frame
S	0.283645	14.4799 ms/frame

TABLE II

MEASURED ACCURACY AND RUNTIME FOR COLOR BASED FEATURES ON THE SPHERE SEQUENCE

Number of candidates	Accuracy	Time
9	0.366971	0.802564 ms/frame
16	0.327784	1.28437 ms/frame
25	0.34285	2.01615 ms/frame
64	0.450254	4.27089 ms/frame
100	0.487683	6.55173 ms/frame
169	0.537917	10.6894 ms/frame
196	**0.541799**	**12.0149 ms/frame**

TABLE III

MEASURED ACCURACY FOR NUMBER OF CANDIDATES USED WITH GRAYSCALE BASED MATCHING ON THE SPHERE SEQUENCE

Step size	Accuracy	Time
0.02	0.553748	12.0736 ms/frame
0.03	0.562598	12.4628 ms/frame
0.05	**0.564566**	**12.5077 ms/frame**

TABLE IV

ACCURACY AND TIME/FRAME FOR DIFFERENT GRID SPACING WITH GRAYSCALE BASED MATCHING WITH 196 CANDIDATES ON THE SPHERE SEQUENCE

Spacing does not effect runtime. For our final result we used spacing 0.05 and achieved 0.5645 accuracy. As we can see from *Figure 2.*, using the grayscale histogram makes sense as the sphere is visibly lighter in the video compared to the background. In the bottom part of *Figure 2.* we can see that the error could be caused by the motion blur due to the rapid movement of the sphere.

car1: we started by testing which feature works best for this sequence. We fixed the number of candidates to 81 and the number of bins to 16. Grid spacing was set to 0.1 as we expected the car to move more in the frame. We tested the different color based features, the results can be seen in *Table 5.*. Using the red color channel performed best. We concluded that testing with over 200 candidates could improve performance but we omitted this because of the performance costs. We then tested for the best grid spacing, our final results can be seen in *Table 5.* Spacing does not effect runtime. For our final result we used a relatively fine spacing of 0.03 and achieved 0.634109 accuracy. In *Figure 3.* we can see that the tracking performance works well even on blurred frames. Our hypothesis is that since we are using the red channel most of the information comes from the location of the rear/brake lights. These remain clearly visible even in noisier frames like the one on top. Our first guess based on the visual appearance was that saturation would perform the best, as the car is decently saturated compared to the background since it is white.

Task 4.4

In this task we are testing the gradient based histogram model for the tracking.

Hyperparameter	Accuracy	Time
Feature		
Gray	0.446427	5.21256 ms/frame
R	**0.499367**	**5.52077 ms/frame**
G	0.443733	4.93072 ms/frame
B	0.327403	4.72041 ms/frame
H	0.0501669	15.1735 ms/frame
S	0.483407	14.2173 ms/frame
Number of candidates		
9	0.40176	0.7259 ms/frame
16	0.426243	1.17126 ms/frame
25	0.439932	1.658 ms/frame
64	0.482568	3.74265 ms/frame
100	0.50854	5.65981 ms/frame
169	0.544186	9.41174 ms/frame
196	**0.553835**	**10.4175 ms/frame**
Step size		
0.02	0.628878	11.4964 ms/frame
0.03	**0.634109**	**10.6164 ms/frame**
0.05	0.604761	10.5847 ms/frame
0.10	0.553835	10.4175 ms/frame

TABLE V

HYPERPARAMETER TESTING FOR THE CAR1 SEQUENCE

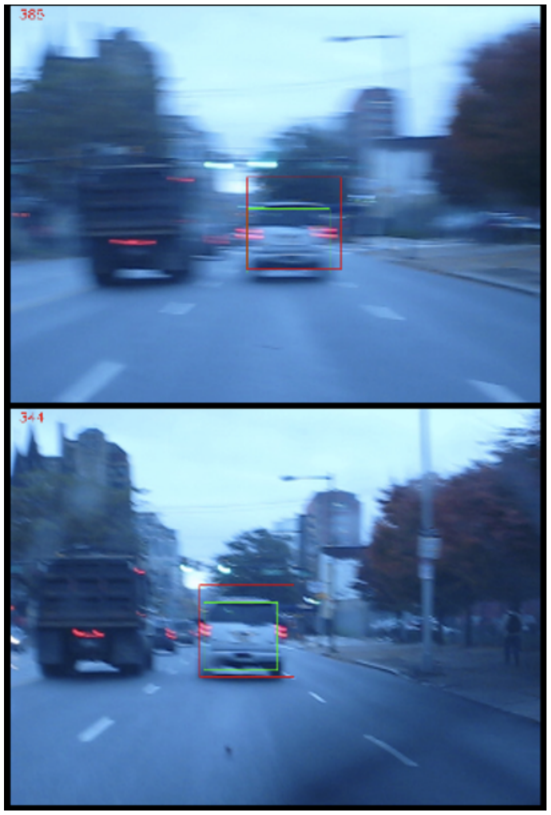

Fig. 3. Example frames from the car sequence

basketball: we started out with number of candidates =25 and conducted a search for the best number of bins. Based on the results we selected 12 bins. Now we fixed the number of bins to 12 and searched for the number of candidates. We selected 169 candidates based on the results. We then searched for the ideal step size between the candidates. The results can be seen in *Table 6.* In *Figure 4.* we can see some common problems during tracking. On the top we can see that while the the prediction is inside the ground truth location the size difference causes an error in the evaluation. In the middle we can see that we find most of Derrick White as his shirt is clearly visible, but probably because of the initialization model the part of the background that is above him on the picture has more similar features than the bottom part. On the bottom image we can see that while the location is generally good, the heavy occlusions from Ron Harper make it imprecise.

ball2: as we expected more movement from the object we started out with 81 candidates and used a step scale of 0.01. We searched for the optimal number of bins. We chose 16 as the number of bins. We then looked for the optimal number of candidates after. We fixed the number of candidates to 196. We wanted to search for the number of bins on a finer scale between 16 and 24. We found that 22 is the optimal number of bins. We finally tested different step sizes, but due to the small size of the object we resorted to a step size of one pixel. The hyperparameter search can be seen in *Table 7.* The main problem in this sequence in our opinion is the size of the object. This would not be as apparent with a color based technique. Another problem is that the ball changes appearance as it rotates and in certain frames *Figure 5.* On the bottom image we can see that due to the size and orientation of the ball the lower bar of the goalpost is picked up as the best candidate.

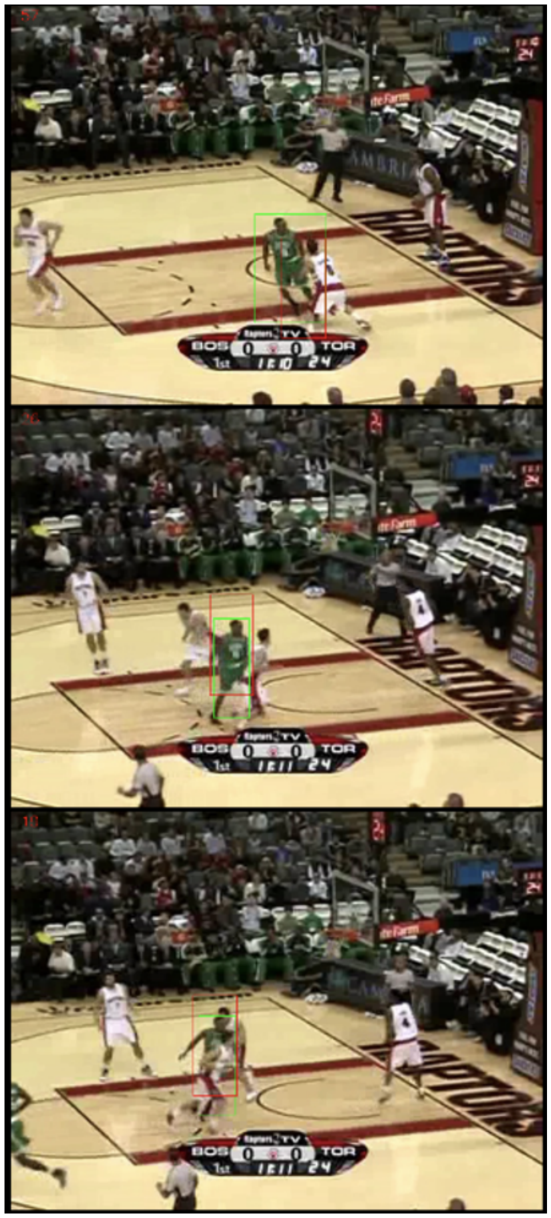

Fig. 4. Example frames from the basketball sequence

V. RUNNING THE CODE

In order to compile, link and run the project, it is only needed run make in the project's src folder and the run the ./main command. We included all parameters used as comments (where we deemed them necessary for convenience).

Hyperparameters	Accuracy	Time
Number of bins		
4	0.0870336	20.9577 ms/frame
8	0.089613	19.3922 ms/frame
12	**0.103203**	**20.7193 ms/frame**
16	0.0901127	20.7621 ms/frame
24	0.0898681	24.9117 ms/frame
36	0.0903394	26.331 ms/frame
Number of candidates		
1	0.115326	1.71172 ms/frame
4	0.181952	4.46007 ms/frame
9	0.0631777	8.25049 ms/frame
16	0.0854546	18.498 ms/frame
49	0.085264	35.0813 ms/frame
169	**0.606101**	**125.487 ms/frame**
196	0.588705	128.023 ms/frame
Step size		
0.02	0.083706	105.666 ms/frame
0.03	**0.617879**	**109.792 ms/frame**
0.05	0.0309896	112.416 ms/frame

TABLE VI
HYPERPARAMETER SEARCH FOR THE BASKETBALL SEQUENCE

Hyperparameter	Accuracy	Time
Number of bins		
4	0.0632237	42.0439 ms/frame
8	0.0617595	42.6863 ms/frame
12	0.0664049	45.8874 ms/frame
16	**0.0683719**	**45.1722 ms/frame**
24	0.0683719	53.7794 ms/frame
36	0.0678998	54.4116 ms/frame
Number of candidates		
1	0.0359241	3.79308 ms/frame
4	0.0382483	14.0193 ms/frame
9	0.0380392	8.16701 ms/frame
16	0.0382483	16.7267 ms/frame
49	0.0611726	36.009 ms/frame
169	0.225094	108.548 ms/frame
196	**0.307662**	**127.96 ms/frame**
225	0.298009	140.304 ms/frame
Number of bins		
16	0.307662	147.941 ms/frame
20	0.307782	143.214 ms/frame
22	**0.31744**	**143.429 ms/frame**
24	0.317524	155.082 ms/frame

TABLE VII
HYPERPARAMETER SEARCH FOR THE BALL2 SEQUENCE

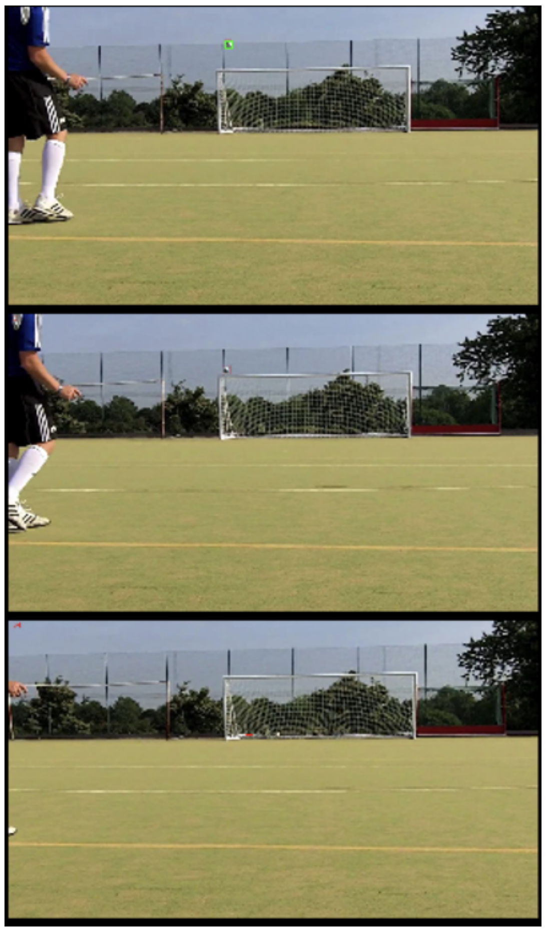
Fig. 5. Example frames from the ball2 sequence

VI. USAGE

VII. APPENDIX

A. Individual Work Reflection

For the measurement extraction the hardest part was finding the correct hyper-parameters. For Kalman-filtering the hardest part was to hard-code the matrices, without making any mistakes in it.

B. Code Merging Feedback

The code merging was smooth since the 2 parts was following the principals of the clean coding, including code encapsulation and good comments. The code was organized in different functions, classes which made the main file cleaner and easier to understand each others code.

C. Time Log

- Individual task: 3-4 (per person)
- Data analysis: 0.5 hours
- Code Merging: 1 hour
- Parameter Tuning and analysis: 6 hours
- Report: 4 hours

Total: 15 hours

D. Pair Programming Reflection

Pair programming technique was useful when we sat together and merged the project from our individual parts. This technique is used in real scenarios as well when software engineers have to combine their differently developed tasks. Parameter tuning might not be done in pair working, instead one person should change the parameters the other person tests the model with this setting.