Applied Video Sequence Analysis
Lab 2 "Object detection and classification"

Hunor Laczkó and Anne-Claire Fouchier

I. INTRODUCTION

This laboratory report presents blob analysis routines, written in C++ using the OpenCV library, that were integrated into a given blob analysis framework. Using the output of OpenCV's background subtraction modules for which the input video sequences are assumed to be captured from a still camera, the routines developed for this laboratory are first to extract blobs using the foreground segmentation mask and the Grass-fire algorithm, eliminate small blobs, then classify them as people, cars or objects. Additionally, a routine for stationary foreground detection was implemented based on the paper mentioned in the references.

II. METHOD

A. Blob Extraction

Blob extraction consists on extracting Binary Large Objects from the given foreground masks. The extraction is based on the Grassfire algorithm and the number of returned blobs depends on the pixel-connectivity strategy. These blobs are then filtered; the smaller ones are ruled out; and the remaining ones will become candidates for classification.

B. Blob Classification

The classification routine is based on the aspect ratio feature and a simple Gaussian statistical classifier. A blob can either be classified as a person, a car, an object, or remain unknown if it does not belong to any of the former classes.

C. Stationary Blob extraction and classification

This method is based on [1] from which the relevant equations can be found below.

First, the Foreground History Image (FHI) is updated with the measure of the foreground temporal variation, considering foreground and background detections as follows:

$$FHI_t(x) = FHI_{t-1}(x) + w^f_{pos} * FG_t(x) \quad (1)$$

$$FHI_t(x) = FHI_{t-1}(x) - w^f_{neg} * (\sim FG_t(x)) \quad (2)$$

Weights are assigned to manage the contribution of the foreground and the background from the foreground mask. The foreground score increases when the pixel is foreground and decreases when it belongs to the background model.

The Foreground History Image is then normalized to the range [0, 1] considering the video frame rate (fps) and the stationary detection time (t_{static}) :

$$\overline{FHI_t}(x) = min(1, FHI_t(x)/(fps * t_{static})) \quad (3)$$

$$SFG_t(x) = \begin{cases} 1 & SFG_{t-1}(x) \geq \eta \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

This method is used to keep detecting foreground objects after they become stationary.

D. Grassfire algorithm

Based on [2] two self developed implementations are provided for the Grassfire algorithm, these two being the recursive and sequential approaches.

III. IMPLEMENTATION

The different tasks are implemented in the same project where the different modes of operations can be controlled with command line arguments. These will be detailed in the Running the code section. All the functions are implemented in the *BlobList* class which in turn uses *Blob* class for storing the blobs. A blob is represented as a bounding box with an id and a class label. The respective functions will be detailed in the next sections.

The *BlobList* class stores a vector of the blobs and operates on that vector. It also contains a *fillMode* variable which determines which function is used for determining a given blob. Two instances are initialized from the class, one for the simple blob operations and one for the stationary background operations. All later operations are called on these two instances.

More precise implementation details can be found in the source files as comments at the corresponding locations.

A. Functions for blob extraction

The extraction of the blobs is done with the *extractBlobs* method, it takes as an input the foreground mask and a connectivity value (this determines the kind of connectivity used, either 4 or 8) and extracts all the blobs from the mask and stores them in the class member *bloblist*.

For the extraction the image is traversed on a pixel level; when a foreground pixel is found, the corresponding blob is determined with the help of *cv::floodFill()* function which is a built in OpenCV function. It uses that found pixel as a seed point, fills the corresponding blob with a given value and returns the enclosing rectangle for the blob which will be used for its representation. This bounding box will be added to the *bloblist* and the process is repeated until every pixel is processed.

After the extraction, a post-processing routine is performed on the bloblist in order to remove the small blobs, considered noise. To do this, the *removeSmallBlobs* method is used.

It traverses the *bloblist* vector and removes all blobs that have width smaller than *min_width* or height smaller than *min_height*, which are the input parameters of the method.

B. Function for blob classification

For classification of the blobs the *classifyBlobs* method is used. It has no parameters since it operates on the class member *bloblist* and the models for the classification are stored globally.

The existing *bloblist* is traversed and each blob's bounding box's aspect ratio is calculated. Then the distance of this value is calculated from the model classes' means. If the distance falls into the interval of [mean - 3 * standard deviation, mean + 3 * standard deviation], it is considered to belong to that class. It could happen that it belongs to multiple classes, in that case the one with the smaller distance is chosen. If it doesn't belong to any of the classes its label will be "Unknown".

C. Function for stationary blob extraction and classification

The function *extractStationatyFG* takes three arguments: *fgmask*, *fgmask_history* and *sfgmask)*. *fgmask* is the foreground mask of the analysed frame, *fgmask_history* is the history of the foreground mask and *sfgmask* is the generated stationary foreground mask.

Using the earlier described formulas a history of the foreground mask is stored and updated at each frame. The update stores the weighted amount of time the pixel spends as foreground, meaning it gets decreased (weighted) if it belongs to the background and increased (weighted) if it belongs to the foreground. This value is later normalized and thresholded which will determine if it belongs to the stationary foreground or not. Once the stationary foreground is determined, the same operations are performed on it as the simple foreground mask.

It has to be noted that, in order to the stationary objects not to get incorporated into the background by the background segmentation method, the learning rate had to be set to 0. Anything above zero, even slightly, would result in the stationary objects to eventually getting incorporated, and thus removed from the stationary background.

The method can be controlled with five constants that can be found on top of the implementation of the code. These determine the positive and negative case weights for the history update, the fps and seconds for the normalization and a threshold value for determining the stationary mask.

D. Function for Grassfire

The Grassfire algorithm can be used instead of OpenCV's *floodFill* implementation to determine a specific blob. There are two different approaches, a recursive and a sequential. Both of them were implemented, in the methods *fillRecursive* and *fillSequential* respectively. Both of them were designed in a way that their interfaces resemble as closely as possible with OpenCV's implementation this way making it easy to interchange them.

The recursive version is made of two methods, a wrapper method *fillRecursive* which assures that the incoming *rect* parameter is centered around the seed point (this way the function is self contained, doesn't depend on the parameter) and the *doFillRecursive* method which does the recursive step itself. In this method a depth-first style traversal is implemented which visits the neighbours of the seed point, then their neighbours and so on. Each step the current pixel's value is changed to signal that it was already processed and the *rect* parameter is updated to contain this pixel too. Once there are no more pixels to visit the recursion will stop and the *rect* parameter will contain the bounding box of the blob. While this method is easy to implement and works fast the recursion might go too deep even on medium resolution images which can cause a stackoverflow.

The sequential method solves the problem of stackoverflow but subjectively it was slightly slower the the recursive version. An explanation for this could be that the pixels to be visited have to be stored in each step. This method resembles a breath-first traversal where in each step the neighbours to be visited are stored in a FIFO queue and in the next step the first element of this queue is processed. This is implemented in the *fillSequential* method. It works in the same way as the recursive version, it starts from the seed point and return the rectangle of the bounding box.

E. Running the code

The code consists of the five source files contained in the src folder and the accompanying makefile. Using the makefile, the code can be easily compiled and then it only has to be run. It can be run by executing *lab2*. The program expects three command line arguments:

1) **FillMode**: Determines the function used to find one blob
   - *opencvFill*: Uses OpenCV's floodFill function
   - *recursiveFill*: Uses the recursive implementation of Grassfire
   - *sequentialFill*: Uses the sequential implementation of Grassfire
2) **BackgroundMode**: Determines if stationary background is calculated or not
   - *noStationaryBackground*: stationary background is not calculated, learning rate for background detection is -1
   - *stationaryBackground*: stationary background is calculated, learning rate for background detection is 0
3) **DatasetPath**: the root folder where the three datasets (as found in Section IV.) are located

The program can also be run with one parameter, that being "help" to see a help message about the arguments and their possible values.

IV. DATA

Method	Dataset	Video sequences
Blob extraction	ETRI	ETRI_od_A
Extraction & Classification	PETS2006	PETS2006_S1_C3 PETS2006_S4_C3 PETS2006_S5_C3
Classification & Stationary foreground	VISOR	visor_Video00 visor_Video01 visor_Video02 visor_Video03

The ETRI sequence depicts two people walking on a trail in the park. The challenge in this sequence lays on the tree shadows darkening half of the frame.

The PETS2006 sequences depict a train station hall, with people walking through the frames. These sequences contain challenges like the high angle, shadows, obstruction and people interacting with objects, mainly suitcases.

The VISOR sequences depict a parking lot with a road behind and in front of it. The cars appearing in the frames have different scales, and become stationary. Objects of several classes also appear in these frames.

V. RESULTS AND ANALYSIS

There was no ground truth available for the dataset so the evaluation was done subjectively. Each task was tested on several videos with different parameters until the best parameters were determined. The results for these can be found in the following sections.

A. Blob Extraction

Using OpenCV's floodfill function, to extract the blobs, resulted in appropriate bounding boxes in the majority of cases as it can be seen in Figure 1. However, the method relies on foreground segmentation, meaning that if that was not correct, the resulting blobs would be erroneous too. As it can be seen in Figure 2 the neck of the person is resembles closely to the background which results in two disconnected components. Therefore, the algorithm identifies them as two blobs. Also, small blobs are filtered, resulting here in excluding the head as a viable blob. However, this is the expected operation. The size filtering helps to remove blobs that result from noise in the segmentation. Leaving the minimum size at (20, 20) had good results in most of the cases.

Fig. 1. Result of blob extraction with floodfill algorithm

Fig. 2. Wrong bounding box resulting from bad segmentation

Another parameter of the method was the connectivity value, meaning if 4- or 8-connectivity (number of neighbours) was used to determine the connected components. In theory this would matter for instance, when there are two people close to each other. In that case, the 4-connectivity should be able to differentiate them more easily. In practice, there were no scenes found in the dataset where the connectivity changed the blob extraction results. It would only make a difference if two or more objects were connected with diagonal neighbouring pixel, meaning they only had a corner of a pixel common, for which there is only a low probability. As it can be seen in Figure 3 and Figure 4 if two people are close to each other they do not get recognized separately, only after they get further apart and are distinguishable in the foreground mask.

Fig. 3. The two people are not recognized separately

Fig. 4. The two people are recognized separately

B. Blob Classification

For the classification two parameters could be defined. First the type of distance measure used and second the multiplier for the gaussian matching.

For the distance measure, the *Euclidean distance* and *Weighted euclidean distance* were tested. In the tested scenes, using one or the other resulted in almost no difference so the normal euclidean distance was chosen for ease of computation.

With the multiplier, the size of the interval could be modified which would determine if the blob in question belonged or not to a class. Taking three times the standard deviation from the mean had the best results in the VISOR dataset. The algorithm was able to identify cars in the scenes as it can be seen in Figure 5. The aspect-ratio-based identification proved easy but could only handle limited amount of cases. For instance, in Figure 6, both cars are classified as objects, not cars. The car on the upper part of the image has turned almost parallel with the camera view, meaning for the camera that it sees more of a square shape than rectangle. Since the car model is more rectangle-shaped, in this case the aspect ratio falls closer to the class of object, which explains why it gets classified as object. There is another situation with the second car, the one closer to the camera. The standard deviation of the class object is twice higher than the one of the car class. When it is multiplied by three, the difference becomes even higher. So when the closer car turns completely sideways to the camera, its aspect ratio becomes even bigger. This means that even though it is closer to the car class, it no longer falls into the interval of the car, only into the interval of the object, that is why it gets classified as an object.

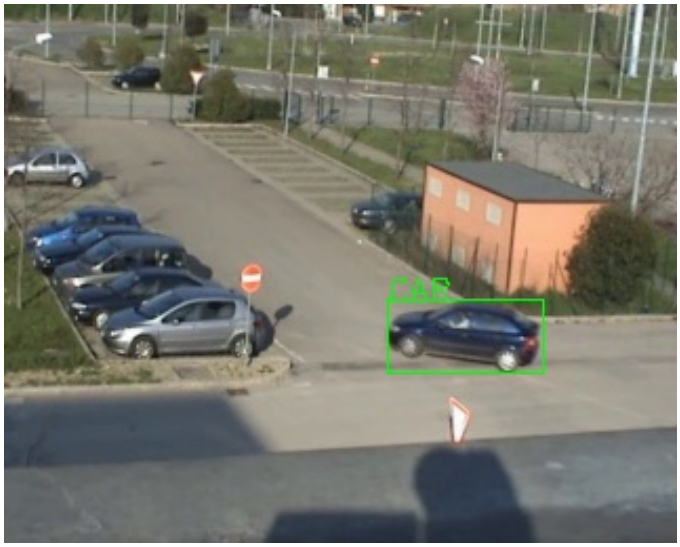

Fig. 5. Object classification result

C. Stationary Blob extraction and classification

Using a zero learning rate helped conserving the stationary objects, but it also introduced a lot of noise. In the ETRI video sequence, a significant number of false detections appeared because of the highly noisy segmentation (probably because of the slight lighting changes). In the VISOR video sequences, the results were better, but in the shadow areas at

Fig. 6. Wrongly classified objects

the bottom, there was some noise too that only got stronger over time and even passed the size filtering so it got falsely detected, as it can be seen in Figure 7.

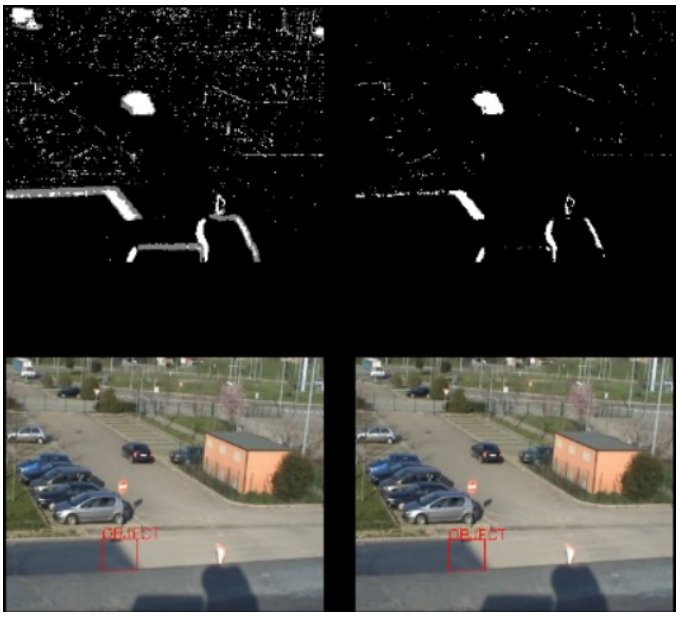

Fig. 7. Noise detected as foreground. The car doesn't get detected because it doesn't pass the size limit.

The stationary foreground detection uses multiple parameters, determining several of its aspects. *FPS* and *SECS_STATIONARY* are used for the normalization; that way essentially determining the interval we consider the history value to be. Using 1 second with an fps of 25 proved to be successful. The history update is controlled by the positive (*I_COST*) and negative (*D_COST*) weights, which determine how much the background and foreground

matter. The most important factor is the ratio between the two weights. If both of them are 1, it means an object has to spend the same amount of time as foreground as much it was background before. To accelerate the algorithm a positive weight of 3 and a negative weight of 2 was used. As a result in *visorVideo00*, the first car that stops is recognized in the stationary foreground after a few seconds of it stopping (Figure 8). The algorithm being sensitive having these parameters, the second car also starts to get integrated, but then, moves again (Figure 9). However, after a few seconds of moving away, it gets reverted back to background (Figure 10). In this situation, using 1 for both weights, would result in the second car not being recognized but the first car would also get recognized way later. The last parameter is *STAT_TH* which is used for thresholding the normalized history. With the weights used, this threshold had no significant effect on the outcome, for the images displayed here, a value of 0.5 was used, but even a threshold of 0.9 worked almost the same.

Fig. 8. First (closer) car recognized, the second (further) has just stopped, so it's not recognized yet

D. Grassfire algorithm

Both the recursive and sequential implementations of Grassfire had exactly the same results as OpenCV's floodfill so there was no visual difference between the three methods. As already mentioned in the implementation section, the sequential implementation seemed a bit slower, but nothing significant. In the end, all three methods performed in real time.

Note: Running the recursive version on the PETS2006 videos may cause stackoverflow depending on the platform (on linux it seemed to be working), which crashes the program. This problem could not be solved and the exact

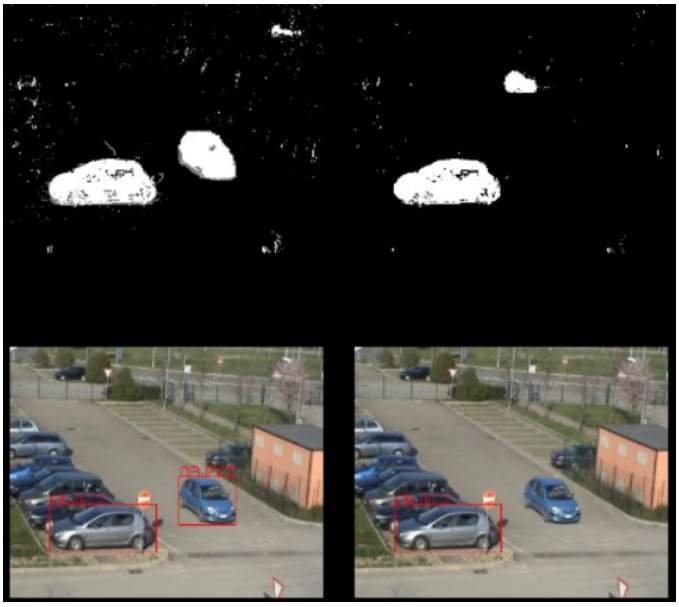

Fig. 9. The second car (further) starts to get recognized but then it leaves

cause could not be determined. On the other videos it worked as expected.

VI. CONCLUSIONS

In conclusion, the program fulfills the requirements. It extracts the blobs correctly based on the foreground segmentation. Considering the simple classifier used, it classifies the objects correctly in the general cases. The stationary foreground is detected correctly and the extraction and classification is also performed on it. Other implementations of the Grassfire algorithm were also considered and tested. In the future, a more complex classifier could be implemented which takes into account more features and the existing classes could be further refined.

VII. TIME LOG

Below the amount of time spent on each aspect of the project is detailed. The times contain both participant's time summed up.

1) Blob Extraction: : 4 hours, the challenge being understanding the OpenCV function floodfill

2) Blob Classification: : 2 hours, finding the right measure and multiplier took time

3) Stationary Blob extraction and classification: : 4 hours, including reading the article and understanding the parameters, especially for equation 1 and 2, debugging the implementation

4) Grassfire algorithm: : 4 hours, implementing both versions and debugging them

5) Evaluation: : 3 hours, taking pictures, finding optimal parameters

6) Report: : 5 hours, assembling and writing

Fig. 10. A few seconds after leaving, the second car's place gets reverted back to background

REFERENCES

[1] D. Ortego and J. C. SanMiguel, "Stationary foreground detection for video-surveillance based on foreground and motion history images," 2013 10th IEEE International Conference on Advanced Video and Signal Based Surveillance, Krakow, 2013, pp. 75-80.

[2] Introduction to Image and Video Processing", Th.B. Moeslund, Springer, 2012