Computer vision: project

Thomas Aelbrecht[1], Andreas De Witte[1], Jochen Laroy[1], Pieter-Jan Philips[1], and Gillis Werrebrouck[1]

Ghent University, Valentin Vaerwyckweg 1, 9000 Ghent, Belgium

Abstract. The abstract should briefly summarize the contents of the paper in 150–250 words.

Keywords: First keyword · Second keyword · Another keyword.

1 Introduction

Write introduction

2 Semi-supervised painting detection

For the unsupervised painting detection, a series of actions are executed. The way this works is as follow. All the contours are being detected by a naive method; however, these contours are not being shown yet. A graphical user interface has been made to make the unsupervised image detection easy and user friendly by providing a variaty of actions. The available actions are the following:

1. Add
2. Remove
3. Draw
4. Drag
5. Convert
6. Clear canvas
7. Save to database
8. Next image

Add action The "Add" action allows the user to click onto the image. The idea is that the user clicks on a painting to be "detect". For this action, the contours that were found at the beginning are being used. When clicking on a painting in an image, the code checks if that click event has been triggered in the bounding box of a detected painting, if so then a rectangle is being drawn on the image. All contours in which the user clicked will be made visible.

Remove action The "Remove" action allows the user to click into visible contours to remove them. This is a necessary action because the algorithm sometimes detects more contours than it should, so these can be removed.

Draw action The "Draw" action has been provided in case the algorithm can't recognize a painting. The user can drag a new contour on the image in case a painting hasn't been detected by the algorithm.

Drag action The "Drag" action makes it possible to drag the individual corners. This is the most useful part of the unsupervised painting detection because the user can adjust individual corners if necessary. The contours need to be converted before the "Drag" action can work. The corners will have little squares on them once the contours are converted.

Convert action The "Convert" action will convert the contours from rectangles into draggable quadrilaterals. This action needs to be executed to be able to drag individual corners.

Clear canvas action The "Clear canvas" action will remove all the visible contours. This is perfect to reset the image to the original state.

Save to database action The "Save to database" action will save all the quadrilaterals that are visible to the database. This action will also display the next image.

Next image action The "Next image" action will display the next image on which all above actions can be performed.

3 The naive painting detection algorithm

The contour detection is the actual logic part. This is the part that decides what contours are in the image and it works as described below.

Before any contour detection can be done, as many unnecessary details as possible have to be removed. OpenCV has several options to solve this problem, for example erode, dilate, blurring (median, Gaussian…) or downscaling. The image needs to get rid of as many details as possible to prevent detection of contours inside the paintings or on the wall. Although this won't be perfect, the removal of details will decrease the number of incorrectly detected contours.

So the first step of the algorithm is to remove details by resizing the image. Our implementation scales the image down and back up with a factor 5. To scale the image back up, pixel values are calculated using the pixel area. The scale up was mostly done to be able to show the original image, but it's also a possibility to use the algorithm with the downscaled image. This will result in a slight increase in performance because the image it's working on would be smaller.

The next step is to convert the image to grayscale. This is done to make it easier to differentiate certain parts. This grayscaled image is then dilated and

eroded to remove even more noise at the borders of the painting or on the walls. The last step of removing noise is to do a median blur over it. The biggest advantage of using median blur is that it will preserve edges while removing noise. The idea behind these steps is to smeare as many details as possible, in other words to make bigger blots with the same color. This makes it easier to detect the borders and remove noise in the background.

The next step is to detect edges with Canny. The result of the Canny function will then be dilated once again to make the found edges stronger. After this the contours can be detected using OpenCV's algorithm. This algorithm will make sure only the most outer contours are returned when a hierarchical structure of contours is found. A small final detail to prevent unlogical solutions is the following: any contour with a ratio smaller than 1:10 will be removed. This prevents very small contours to appear around noisy parts of the image.

3.1 Strengths and weaknesses

This biggest strength of this algorithm is that it will give an output in almost every image. An example of an image where no painting will be detected is an image where the framework of the painting is not visible, meaning the image only contains the painting itself, no wall and no framework. With this kind of paintings, the desired sulution is a contour containing the entire painting, while this is not possible because the painting has no border at all. Another advantage is that the algorithm is very fast in detecting contours due to the fact that all the parts of the algorithm are standard functions that have a good performance.

However, the biggest weakness of this algorithm is that almost every found contour is not precise enough, so almost every solution needs a slight adjustment. For some solutions, the algorithm detects contours that are slightly larger than the actual painting, while for other paintings, it doesn't even include the border of the painting.

Another weakness of the algorithm is that paintings sometimes have overexposure or shadows, which makes it harder to detect it's contours. Some paintings even have a small tag with a description next to the painting, sometimes this small tag is detected as being a painting. This is a logical decision, because there's a big contrast in colors and the tag has a clear contour, but this is not a desired effect.

This algorithm is a first version to quickly be able to fill the database with the groundtruth. The actual algorithm in its final version is completely different and doesn't have these issues anymore. float

4 Unsupervised painting detection

The algorithm in assignment 2 drastically differs from the algorithm in assignment 1 (see Section **??**), it was designed from scratch with the weaknesses of the previous algorithm in mind.

The first step in the algorithm is a mean shift segmentation (see **??**)). This is a method to remove noise by taking the mean of the pixels within a certain range. A big advantage is that it partialy removes color gradients and fine-grain textures which would cause issues in the further steps of the algorithm.

Fig. 1. Mean shift segmentation.

A common technique in object detection is by creating a mask of the object. The problem with this is that we aren't certain about the size and ratio of the paiting as well as where the painting is located in the image and how many there are in the image. This can be solved by thinking the other way around. The one certainty that is consistent throughout all images is that all paintings hang on a wall. A mask for the wall will be made instead of making a mask for the paintings. The mask of the painting(s) can then be obtained by simply inverting the mask.

Another issue is that this mask can't be statically programmed in code for each image since we wan't to be able to use the algorithm on any image or video frame. This can be solved by using a technique called flooding. Flooding will create a mask from a starting position in the image and will fill all neighbouring pixels if they have a color close to the color of the starting position. This is done

recursively until no pixels are withing the color range of the starting position anymore.

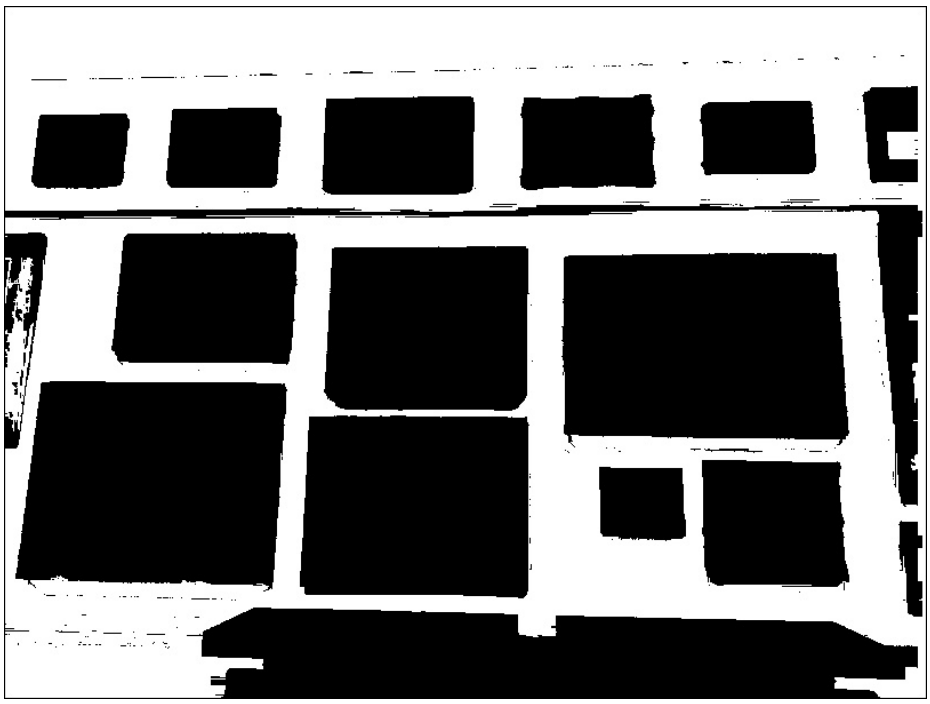

Fig. 2. Wall mask.

The next problem is that there needs to be a good starting position to have a good and correct mask for the wall. To find this, the unsupervised algorithm will iterate over all pixels of the image with a certain step size and perform floodfill with that pixel as starting position. Once a mask is found that has the same height and width as the image, then this is the mask that will be used. If no such mask is found after iterating over all pixels (with a certain step size), then the mask with the bigest size is used. This is because sometimes part of the floor or other elements in the image will obstruct the floodfill algorithm to find a mask that has the same size of the image. See **??**) for an example of such mask.

Once the mask of the wall has been obtained, it is inverted to have the mask of the paintings (see **??**). The mask is then eroded to remove small imperfections in the mask and a median blur is used to smooth the edges of the mask.

The next step is to use Canny, the difference with the naive approach is that the two threshold values are determined by using the Otsu algorithm to choose the optimal threshold value.

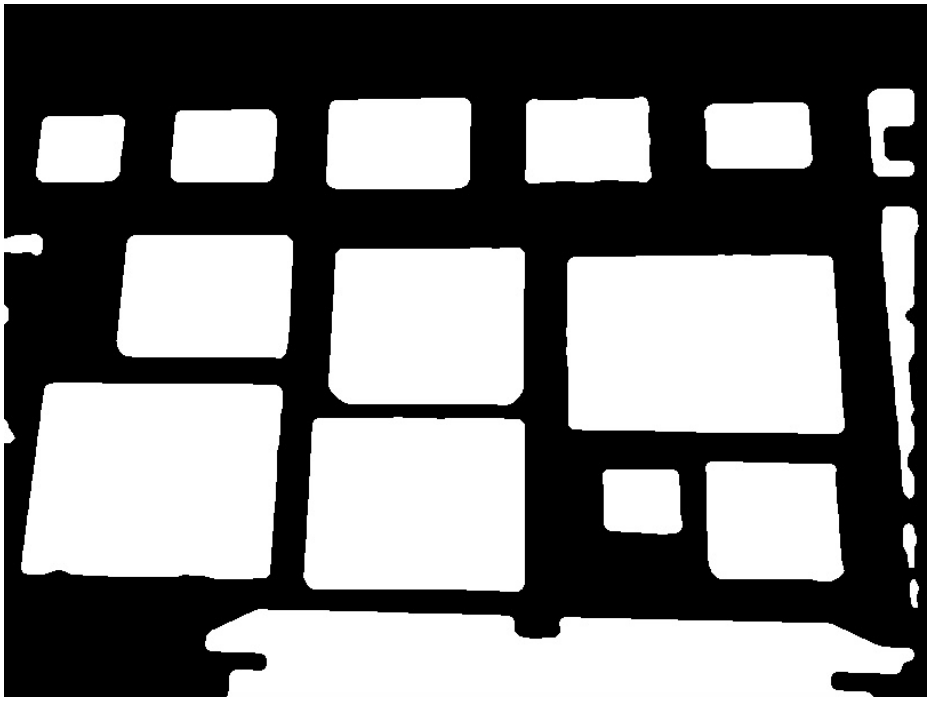

Fig. 3. Paintings mask.

Next, a morphological transformations is performed on the Canny edges (see **??**). This will close edges that are close to each other as this will improve the detection of closed contours. This morphological transformations is in essence a dilation followed by an erosion.

The next steps are exactly the same as the naive painting detection algorithm as described before. The contours are detected in the Canny edges and only fully enclosed polygons with 4 sides are returned as quadrilaterals.

4.1 Strengths and weaknesses

The most important improvement of the detection algorithm is that it isn't based on smearing out colors and finding contours in it anymore. The key to success with this algorithm is the use of automatic mask creation by using floodfill as main technique. Most of the paintings are detected in most of the images. The detected paintings also have a better fit compared to the naive algorithm.

Most paintings in the dataset are correctly detected as well those in the test set. Only on some occasions there is a miss detection. One flaw that has been found is that some doorways are detected as a painting. This is logical because of how the mask creation works with the floodfilling. In the case of such a detection

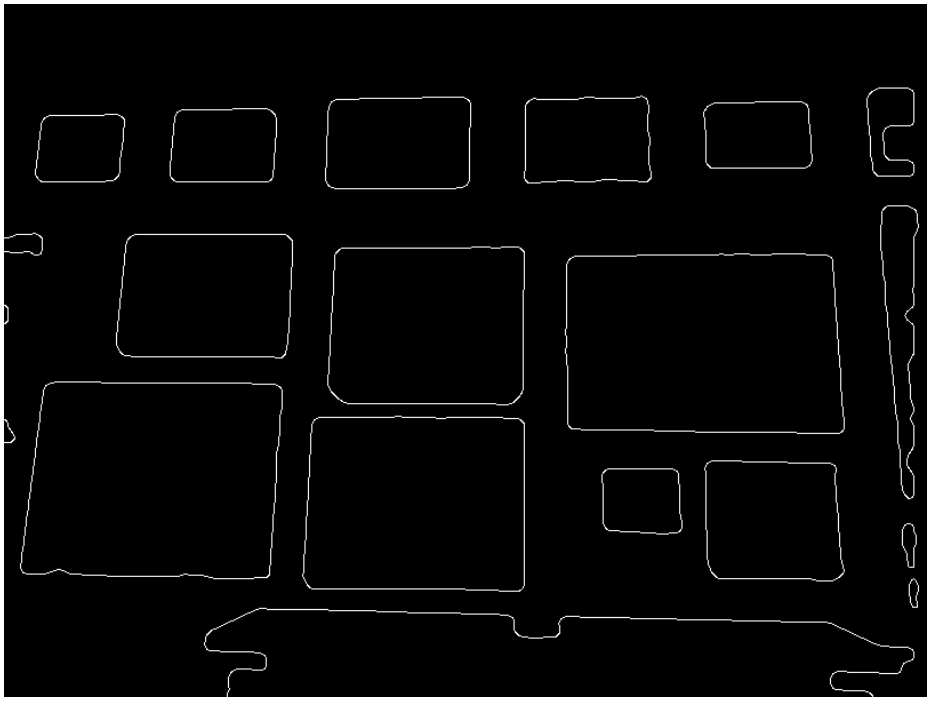

Fig. 4. Paintings edges.

flaw, the doorways obstructs the floodfilling algorithm to correctly create a mask of the non-painting area. This is something that doesn't happen too often.

A minor problem that the algorithm has, which is also the case in the naive algorithm, is that it sometimes detects dark shadows as part of the painting. Although, this is less of an issue in this algorithm then it was in the naive algorithm. It doesn't affect the accuracy too much because if there is a shadow, it only appears on one side of the painting and it doesn't reach far.

A drawback of this algorithm is that it is less performant than the naive algorithm. This is mainly because of the mask creation, specifivally the floodfilling. There has been a slight performace gain by using threading for this but it still remains slower. A possiblity to resolve this and speed up the algorithm would be to use GPU processing power to perform the mask creation. NVidia for example has Cuda cores on which algorithms can be preformed many times faster than on CPUs. This is not possible for this project because OpenCV (developed by Intel) in Python doesn't have the possiblity to execute on GPU's.

5 Quantitative comparison

In order to make a quantitative comparison, two things are needed. First of all, the quadrilaterals have to be found autonomously (as described in Section **??**).

The second thing needed is the groundtruth of the dataset which has been generated by using the solution created in assignment 1 (see Chapter **??**).

To measure the accuracy of the painting detection algorithm, 3 things are required; the amount of false negatives (= paintings that are not found at all), the amount of false positives (= detected paintings that aren't paintings) and the bounding box accuracy (= average intersection divided by union).

With the creation of the solution for this problem, a new problem occurs: how to find the intersection of these two shapes? The solution to this problem is made by using "Shapely". To do so, the quadrilaterals have to be transformed into a polygon. Once this is done, "Shapely" can compute the intersection. It has been tested whether this gives correct intersections if two polygons are not intersecting, or when they are sharing only a line. Also some more general intersections have been tested. Once the intersection is made, it's easy to find the area of the polygon, using "Shapely" once again.

For each of the presumably perfect paintings, all of the found quadrilaterals are checked. The intersection is made and when the "intersection divided by union" is bigger than the previous maximum, than a new best match is found. In the end, when all found quadrilaterals have been tested, a check is done whether the area of the intersection is existing. If this is the case, this value is added to the average intersection divided by union parameter and the amount of found paintings is incremented. If this was not the case, than a painting that needs to be found was not found and the amount of false negatives is incremented. After all of the paintings from the database are checked, the average intersection divided by union parameter is divided by the amount of paintings found. Also, the amount of false positives is calculated by subtracting the amount of paintings found from the amount of quadrilaterals.

The accuracy results for the detection algorithm were fairly good. An example of the detection can be seen in **??**. Red is the detected contours and blue is the groundtruth as determined with the use of assignment 1. The dataset consists of 553 images which all together contain 836 paintings. The detection algorithm had 68 false negatives (paintings) and 35 false positives (paintings). The average bounding box accuracy is 82.80%.

6 Qualitative evaluation

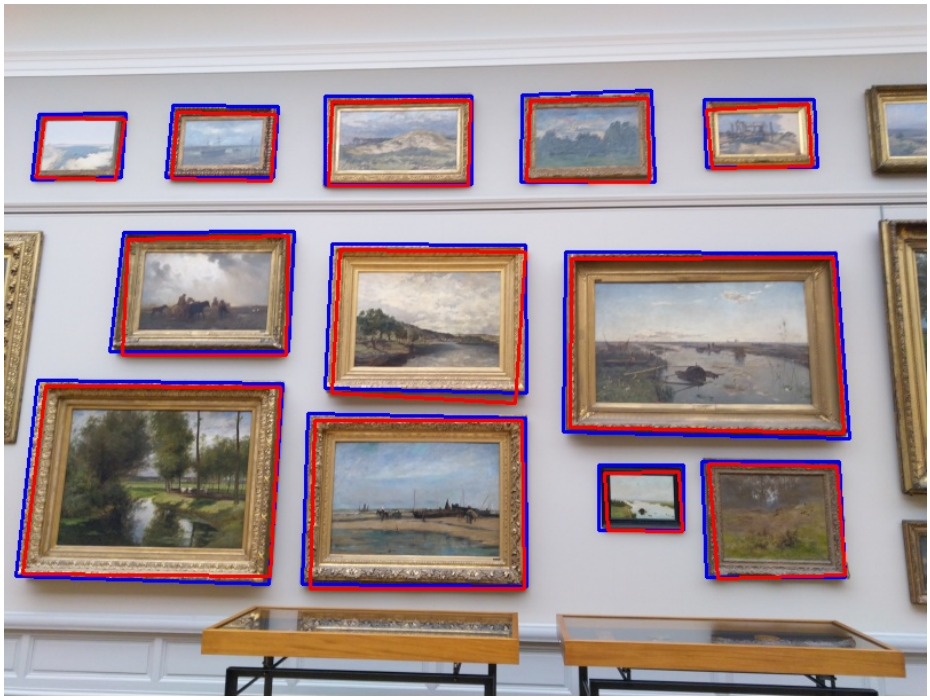

Fig. 5. Paintings detection.