# 1 Introduction

# *Detection, Rectification and Localization of Paintings and*

# *People in a Museum*

|  |  |  |
| --- | --- | --- |
| Laura Valentini  215090@studenti.unimore.it | Luca Casarotti  215059@studenti.unimore.it | Werther Berselli  211507@studenti.unimore.it |

*University of Modena and Reggio Emilia*

July 1, 2020

This paper aims at presenting an overview of the complete pipeline used to detect, rectify from the point of view of the perspective, and identify paintings given an input video filmed inside a museum. The work presented was accelerated using tools such as CUDA and the OpenCV library.

# 2 Related Works

This paper deals with the topic of painting detection and rectification. Neural networks have been used to approach the detection task: similar applications used neural networks to work with paintings from different authors in order to automatically recognize the authors’ style[1] in different works of art.

YOLO V4 [2], a powerful object detection model enabling anyone to train a fast and accurate object detector, has been used to train the painting and people detection model. It is generally used for real time detection in a wide number of applications [3] being fast, accurate and highly customizable.

The OpenCV 4.3[4] library, with its innumerous capabilities in relation to image processing, has been used multiple times throughout the project, especially to extract geometry features.

SLIC, *simple linear iterative clustering*, is a powerful algorithm which adapts a k-means clustering approach to efficiently generate superpixels [5]. In this project, its use was fundamental in order to segment the image and separate the painting from its frame. Its advantages include the simplicity and efficiency compared to similar methods.

K-means is a useful method for clustering images, used in various computer vision applications. In this paper, its use was exploited by the implementation of the SLIC algorithm. Another good example of implementation is aimed at shape classification through deep learning models [6].

In this paper, the Hough transform have been used to detect various kinds of shapes during the rectification phase: despite being pretty dated, it is still an effective algorithm used for lines and shapes detection in images, counting innumerous implementations [7].

Many methodologies have been implemented to extract valuable features for the rectification task: the Canny edge detector [8] was especially useful to ease the definition of paintings contours in each video frame. This detector is generally used in preprocessing phases, before applying other tasks defined in a well-organized pipeline.

The DHash algorithm have been used to compare the rectified paintings images with the provided database entries. A different application for this algorithm was successfully implemented by forum developers with the aim of detecting prohibited content uploaded as profile pictures by its users[9].

# 3 Pipeline

The program presented in this paper takes as an input a video taken inside a museum and performs several steps in order to detect and identify paintings and people, processing each frame and working with an existing database containing paintings and related information. A GUI simplifies the user experience, granting more options to make the use of the tool as flexible as possible. The results are shown via console and a preview of the processing is shown to the user throughout the job.

In the following paragraphs each step will be presented and explained more in detail.

## 3.1 Paintings and people detection

The task including the painting and people detection has been accomplished by training a model.

Firstly, in order to obtain enough data for the training process, a tool called *video\_bbox* has been used to label videos obtaining around 40’000 annotated images containing people and paintings. The resulting dataset has then been reduced up to 9’000 images by skipping 4 continuous frames every 5.

A network able to recognize paintings and people was trained using YOLO v4 on a GTX 1080 single GPU within almost 15 hours. The process resulted in 9100 iterations with a maP of 87.09%. At the end of the training, the following loss graph was obtained.

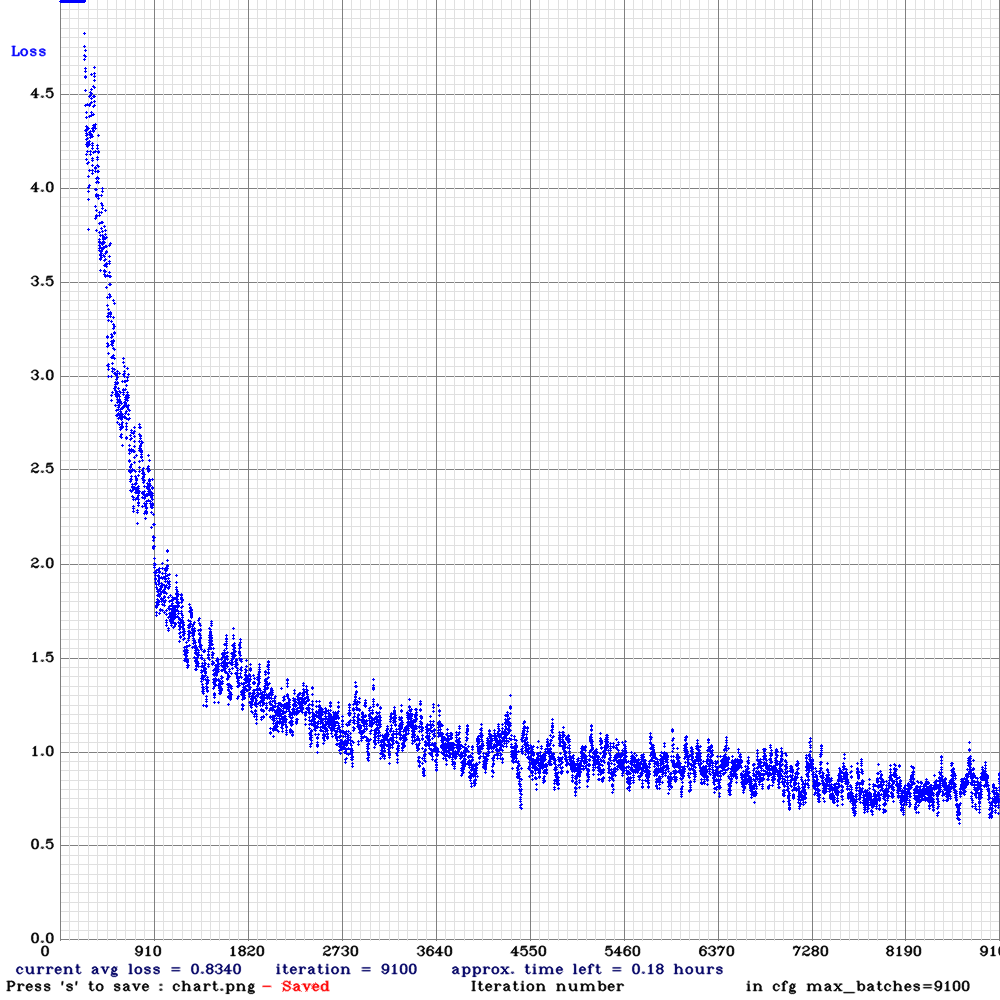


Figure 1: Loss graph of the training process.

The detection process outputs for every painting or people detected the coordinates of the related bounding box.



Figure 2: Detection preview.

## 3.2 Painting rectification

The rectification task works with an image containing a detected painting and the data related to its bounding box.

To rectify each painting detected in a frame of the video, the first step is that of cutting the image to the dimension of the bounding box found in the previous point. Then, with the ROI image, two different approaches are used to find the right points for the rectification of the perspective.

The first approach (*rectification\_polygon*) has the main goal of searching for polygons, parallelograms, in the ROI image. These areas, if found, are highlighted since they hopefully represent the border of the painting without its frame. This shape can then be used to find the necessary points for the homography to be computed.

The pipeline is the following:

* Firstly, two parameters, *alpha* and *beta*, are used to change the luminosity and the contrast of the ROI image, following the formula alpha \* src\_img + beta clipping the range between 0 and 255.
* The resulting image is then converted to grayscale using OpenCV.
* A third parameter is used to apply a threshold on the gray image. Few tests showed how the option which worked best with most of the images was a binary inverted threshold. Many combinations of these three parameters are used in this project in order to work efficiently on the majority of the cases.
* Contours are then searched inside the thresholded image with the OpenCV function findContours. For each contour, the area and the up-right bounding rectangle containing it (specifically its x, y, w and h coordinates) are computed. Every contour having an area lower than 10000 pixels or containing the entire image is discarded.
* The OpenCV function approxPolyDP detects regular polygons and moves forward with the algorithm only in case of a match with a four-sided polygon. In that case, the shape is drawn on a white image with the same dimensions as the original one.

The second approach (*rectification\_mask*) used to find these four points aims at exploiting segmentation.

The pipeline is the following:

* Firstly, K-Means is applied to do a first segmentation to enhance the image. For this project, it’s been used a number of clusters equal to 10, which proved out to be enough to reduce brightness’ variance in the image.
* The segmented image is then converted to the Lab color space in order to use the OpenCV function createCLAHE to increase the contrast.
* SLIC is applied to divide the painting from its frame. This is the crucial part of the approach, as well as the one more reliant on the parameters. Theoretically, the number of superpixels to use in this step would be 2, one for the frame and one for the painting, but this number of superpixels after some tests turned out to be a constraint too tight, especially because in some videos, the painting’s color is too similar to frame’s color, resulting in a single superpixel for the entire image. To solve the problem, the number of superpixels has been increased to 4 which, qualitatively, gave the best results on tests on some more critical videos, proving to be able to separate correctly the painting from its frame. The compactness (parameter used to balance color proximity and space proximity) has been set to 5, while sigma (the width of the Gaussian smoothing kernel for pre-processing) to 1. Both parameters were chosen not to be too tight, in order to generalize as much as possible the segmentation. After this process, the image is converted back to the original color space (BGR).
* The image is now converted to grayscale and a bilateral filter is applied to smooth the edges.
* Otsu thresholding is then applied to the smoothed image and Canny is used to detect the painting’s edges. A closure is also performed to avoid the detection of small regions inside the painting as a polygon;
* The OpenCV function findContours , and draw a mask that covers the width and height of the contour found (mask) which is then applied to the original image to find the four points (mask’s corners) to rectify the image.

To avoid exceptions, a second mask (a grayscale image with a value of 255 everywhere, except for the borders, where it is 1) was added to prevent the case in which by drawing the mask the rectangle would reach the borders of the original image. The result is the logical AND between the two masks.

Both methods return an RGB image with a visible highlighted area indicating the shape of the painting without its frame.

A method called findCorners() is then applied to this image, detect the corners of this area. It converts the image to grayscale and, thanks to the OpenCV function goodFeaturesToTrack(), detects four points, draws them on the initial image and returns them. These four points are, first of all, ordered such that the entries in the list follow this syntax [top-left, top-right, bottom-right, bottom-left].

The four ordered points are then used to compute the width and height of the new image with which it is possible to construct the set of destination points to obtain a “birds eye view”. Now, it is possible to compute the perspective transform matrix and apply it to warp the image thanks, respectively, to the OpenCV functions getPerspectiveTransform and warpPerspective.

Although less precise than the previous one, which works smoothly on easy cases, such as rectangular paintings, the second approach can work with a higher range of images with the same set of parameters, it rectifies correctly paintings with a large distortion or lateral views as well as a wider range of shapes.



Figure 3: Detection and Rectification preview.

## 3.3 Painting retrieval

The retrieval task works with the cut and rectified painting and outputs, in case of a match with the given database, the data related to the most probable result.

Many solutions, such as a first approach with ORB, have been implemented and compared for the retrieval task. The guiding principle of our analysis was to find a fast and accurate approach, given some performance constraints by design. Considering this, the DHash algorithm was implemented.

This algorithm works by reducing each image to a 9x8 size as well as reducing its color scale. The differences between pixels and their nearest neighbors are then computed and, based on those results, the algorithm computes an hash.

In the setup phase, all the hashes related to the images contained in the database are calculated, and for each ROI, the corresponding hash is computed and compared to all the database stored hashes, using a distance measure. If the minimum value is below a predefined threshold, the corresponding painting data are retrieved and printed on console.

This algorithm turned out to be really fast and efficient on every test performed, with a good detection accuracy.

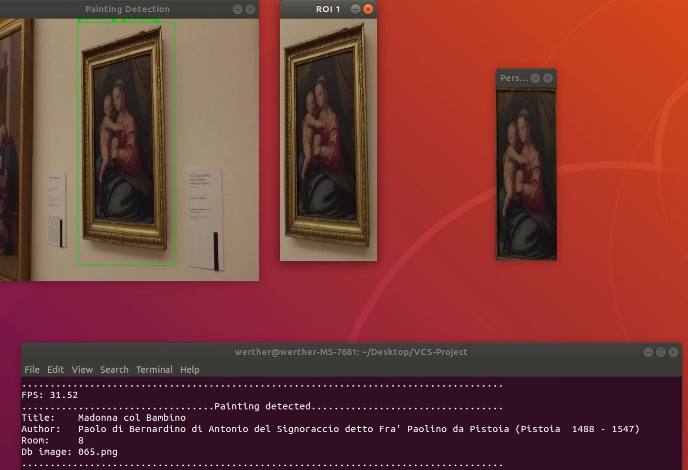
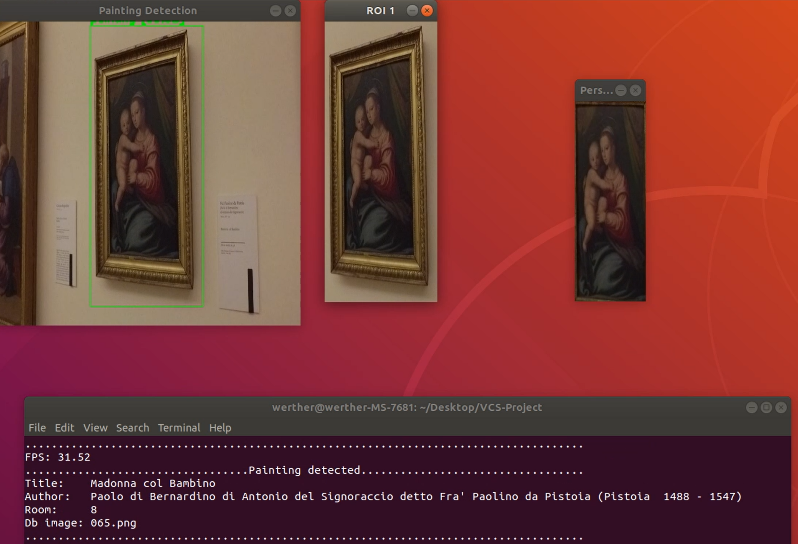


Figure 4: Painting retrieval and *room\_id* output.

## 3.4 People localization

The localization task has been achieved within the painting retrieval task. After detecting all the paintings inside a frame, if a painting is also found inside the database the saved *room\_id*  is saved and every person detected in the same frame is assigned to that room. The information is the printed on the console.

## 3.5 People facing a painting

The number of people facing a painting in each frame analyzed has been acquired using the information retrieved by placing side by side a pre-trained face detection network with the already trained people detection network. In particular, for each frame, the algorithm computes the number of people and the number of faces in the scene and subtracts them.

People whose face is hidden are considered likely to be facing a painting. The resulting number is printed on console.

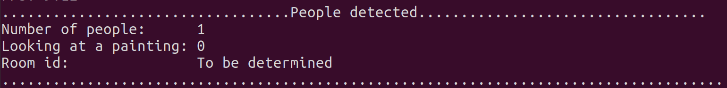
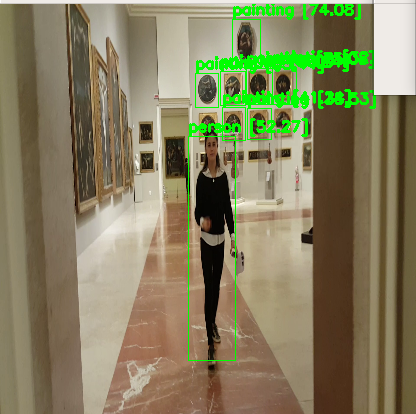


Figure 5: People facing a painting output.

# 4 Additional features

Some additional features have been added to make the user experience more flexible.

First of all, a *Graphic User Interface* has been implemented to make the application as simple to use and configure as possible. Through the GUI it’s possible to customize some parameters related to the computation. These options do not include rectification parameters, which might be tricky to understand and choose not knowing what they stand for in the code.

A “skip” factor is present which allows the user to reduce the total video computation time skipping *n* frames after each processed frame. It is highly useful with long videos.

It is possible to pause the video computation at any time pressing the <space> button to allow the user to stop and analyze the progress made so far. Pressing <space> again will resume the computation. This feature provides more control over the processing but is available only by starting the application with sudo since it requires admin privileges.

If the computation is too chaotic, there is an option on the GUI allowing to set an FPS limiter to slow down the process.

It is also possible to produce an output video rendered using painting detected frames, which can be later reproduced when the process is done.

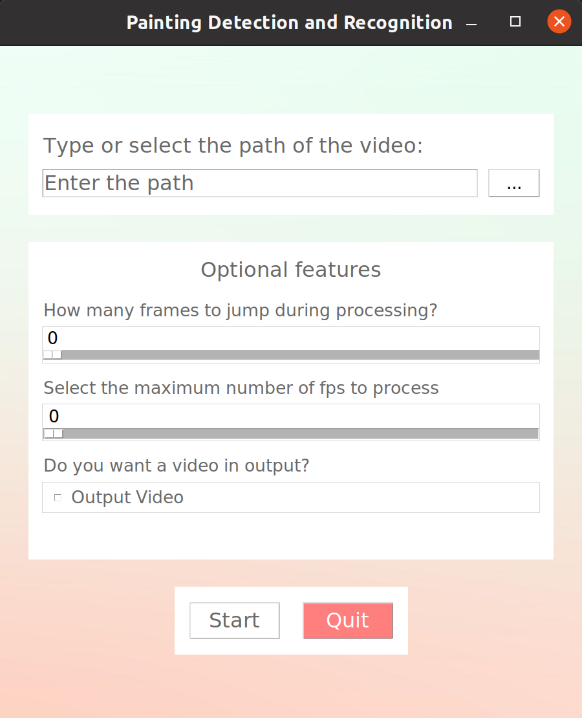


Figure 6: GUI and optional features.

# 5 Results

Some qualitative measurements have been made to test the overall quality of the application.

## 5.1 Accuracy

It was not possible to test with a precise measure the total accuracy, accounted for each singular task. In a qualitative way, every task performed well with the majority of videos.

Some tested videos include VIRB0392.MP4 and 20180206\_114720.mp4.

## 5.2 Timing

The application has been tested on three different systems, all CUDA-enabled:

1. Laptop with I7 9850H, GTX 1650
2. Ryzen 3700X, GTX 970
3. I7 2600K, GTX 1080

Average timings are calculated for two different scenarios: a much simpler video containing only two paintings (VIRB0392.MP4, 34 seconds) and a more complex video with 30 paintings and 4 people detected (20180206\_114720.mp4, 26 seconds).

Tests have been performed both separately on each task and all together to better understand the weight of each task in the total process.

### **Simple Video**

The painting detection is the only task which can actually benefit from CUDA acceleration. In that case, the GTX 1080 showed the best performances. For every other task, the fastest Ryzen 3700X performed better than the other two systems.

In the total frame time overheads and cv.imshow() times were taken into account.

In the simple case, there is a maximum of only two paintings per frame which prevents the Others field from affecting excessively the total time (~10%).

|  |  |
| --- | --- |
|  |  |
|  |  |

Figure 7: Time performance on a simple video.

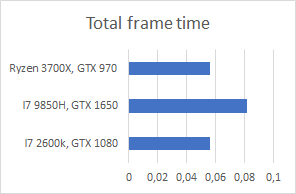


Figure 8: Total Frame Time on a simple video.

### **Complex Video**

In the complex case every task turned out to be way heavier, as expected. In this case, the Others field affected significantly the Total frame time. This consequence is due to the large number of cv.imshow() calls, several for each frame but necessary to show an explanatory preview to the user.

|  |  |
| --- | --- |
|  |  |
|  |  |

Figure 9: Time Performance on a complex video.

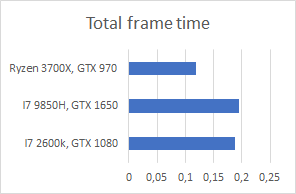
****

Figure 10: Total Frame Time on a complex video.

# 6 Conclusions

The detection and rectification tasks have been tested with a variety of data coming also from other museums and devices: they both responded well to the majority of the videos analyzed. The retrieval task performed incredibly well on the inputs tested, but the shortage of data in the database made it difficult to jump to accuracy conclusions.

In order to improve the speed of the application a “Hide preview” feature could be added: in the most complex computations this option could lead up to a speedup of 2.

# References

1. Adrian Lecoutre, Benjamin Negrevergne and Florian Yger. Recognizing Art Style Automatically in painting with deep learning. JMLR: Workshop and Conference Proceedings 80:1–17. May 2017.
2. Alexey Bochkovskiy .: Yolo-v4 and Yolo-v3/v2 for Windows and Linux. 2019.
3. [Alexey Bochkovskiy](https://arxiv.org/search/cs?searchtype=author&query=Bochkovskiy%2C+A), [Chien-Yao Wang](https://arxiv.org/search/cs?searchtype=author&query=Wang%2C+C), [Hong-Yuan Mark Liao](https://arxiv.org/search/cs?searchtype=author&query=Liao%2C+H+M). YOLOv4: Optimal Speed and Accuracy of Object Detection. *arXiv.*2020.
4. Bradski, G., 2000. The OpenCV Library. Dr. Dobb’s Journal of Software Tools. 2000.
5. R. Achanta, A. Shaji, K. Smith, A. Lucchi, P. Fua and S. Süsstrunk. SLIC Superpixels Compared to State-of-the-Art Superpixel Methods in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 34, no. 11, pp. 2274-2282, Nov. 2012. doi: 10.1109/TPAMI.2012.120.
6. Swe Zar Maw, Thi Thi Zin , Mitsuhiro Yokota and Ei Phyo Min. Classification of shape images using k-means clustering and deep learning. ICIC Express Letters, vol.12, no.10. May 2018.
7. Richard O. Duda and Peter E. Hart. Use of the Hough Transformation to Detect Lines and Curves in Pictures. (PDF). Artificial Intelligence Center. April 1971.
8. J. Canny, A Computational Approach to Edge Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679-698, Nov. 1986, doi: 10.1109/TPAMI.1986.4767851.
9. Hackerfactor. (2013, January 21). Kind of Like That [Blog post]. Retrieved from

http://www.hackerfactor.com/blog/?/archives/529-Kind-of-Like-That.html