# Editing images using hand gestures

A suitable system for editing images with simple gestures based on OpenCV.

Andrea Lo Russo

Department of Computer Science and Engineering
University of Bologna, Italy
andrea.lorusso6@studio.unibo.it
0000925492

2021/2022

**Abstract**

The development of this system is based on the intention to combine computer vision and image processing techniques. Basically, the system allows to modify images using gesture. This means that computer vision techniques are applied to make the gestures, while image processing techniques are applied to edit the images. However, all the techniques used are implemented thanks to the openCV library.

The following report explains in detail how the whole system was implemented. The description is divided into three chapter.

In the initial section of the first chapter, some simple preliminary operations to prepare the working environment are described. In the subsequent sections, all computer vision operations are described as well as the two main tools that enabled the creation of the gestures, namely convexity defects and principal component analysis.

The second chapter explains how the tools described in the first chapter are combined to create gestures.

The last chapter explains all the interactions that allow the user to navigate through the system using gestures, and then describes how gestures allow changes to be made to the images.

# Contents

# 1 Overview of used tools

In this first chapter, all the preliminary operations and tools used to implement gestures are explained.

## 1.1 Preliminary operations

This section describes some preliminary operations necessary to prepare the environment. These operations are implemented on the frames, which are firstly captured by means of the object VideoCapture. The first preliminary operation consists of horizontally flipping the frames through the function cv2::flip. This creates a mirror effect which simplifies the usage of the gestures.

Secondly, a Region Of Interest (ROI) is identified. The region is delimited by a square created by the cv2::rectangle function and indicates where to place the hand [Figure 1]. A clarification is necessary at this point: the ROI has static dimensions, it does not change dynamically according to the resolution of the used webcam. Specifically, the system has been tested on a webcam that has a $480 \times 640$ resolution. This means that, because of the ROI's static nature, it could not work with different resolutions.

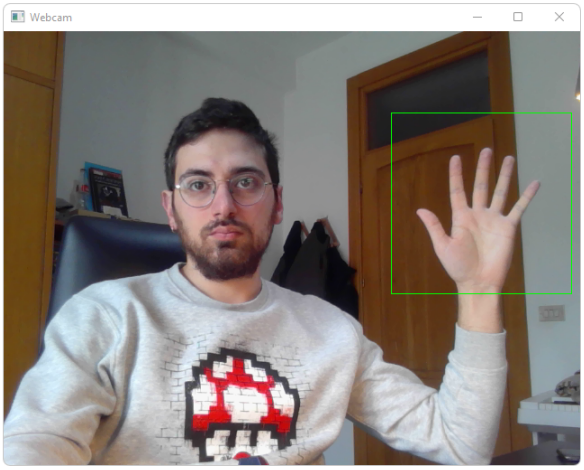



**Figure 1:** Hand placed in the identified ROI.

Thirdly, the function cv2::putText was used to describe the current state of the system. The text is displayed on a section concatenated to the various frames. We will refer to this section as 'caption'.

## 1.2 Segmentation

Once the ROI has been identified, it is segmented to detect the hand with which to make gestures.

The OpenCV library uses the BGR (Blue, Green and Red) model as a standard to represent the colorspace. Since colors are coded using these three channels, only the information about the color is used to segment. Hence, this kind of segmentation is not particularly effective. A more efficient model is the HSV model: colors are represented according to color tone (Hue), degree of Saturation and intensity (Value).

Consequently, the first step for proper segmentation is to convert the ROI frames to the HSV model via the cv2::cvtColor function. Segmentation is then achieved through the use of the function cv2::inRange, which has three arguments:

- A first input array (the ROI)
- An inclusive lower boundary
- An inclusive upper boundary

In this case, as colour based segmentation is being used, also the boundaries are arrays. The purpose of the function is to check whether the values of the elements of the source image lie between the values

defined in the lower and upper bounds. The elements of the source image that respect the constraint will take the value [255, 255, 255] representing white in the corresponding elements of the destination array, while all other elements will take the value [0, 0, 0] representing black. The result is a binary image.

The segmentation obtained in this way presents some problems:

1. the result is susceptible to lighting conditions, especially in an uncontrolled environment.
2. the objects in the background may fall within the chosen segmentation range, incorrectly assuming the value [255, 255, 255].

The first problem described causes mostly impulse noise. There are several approaches to solve it. In my case, applying a median filter, using the function cv2::medianBlur, proved to be very effective. The result obtained was then smoothed slightly by using the function cv2::GaussiabnBlur to improve any imperfections. However, in an uncontrolled environment, lighting conditions can vary over time, making this solution ineffective. In order to consolidate its effectiveness and at the same time solve the second problem - that is the segmentation of objects other than the hand - I decided to set the lower limit and the upper limit of the function cv2::inRange in such a way as to detect only the Blue colour, which is easily captured by a webcam without causing excessive noise. For this reason, I wore a blue glove to use the system [Figure 2].

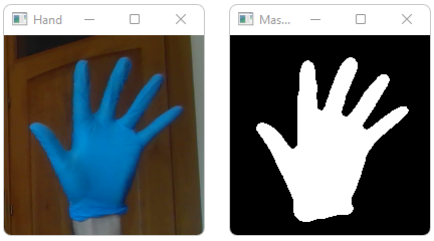



**Figure 2:** Result of segmentation using a blue glove.

The result of the segmentation thus obtained allows the entire system to be used without any particular problems. Moreover, this solution was also preferred because estimating a reference colour such as skin colour can be complicated. Indeed, it varies not only according to environmental conditions but also from person to person. Also, a control panel has been implemented to calibrate the reference colour in a flexible way.

Typically, the segmentation is followed by the connected components labeling. In this case, this step is unnecessary, because the object of analysis, i.e. the hand, is only one.

## 1.3 Edge detection

The system uses two main operations to implement gestures:

- finger detection, obtained by using the function cv2::convexityDefects [section 1.6];
- hand inclination, obtained using cv2::PCAcompute2 [section 1.7].

Both the functions will be detailed in the successive sections and require contours. That's why the hand segmentation is followed by the edge detection, implemented via cv2::findcontours function. For better accuracy, it is suggested to use binary images. For this reason, ROI segmentation precedes contour detection: indeed, it not only isolates the hand as previously described, but also provides a binary image as output. cv2::findcontours function identify changes in intensity between objects and background, so it stores the (x,y) coordinates of the boundary of a shape with the same intensity, which represent the contour.

### 1.3.1 How to find contours - cv2::findContours

The function has three arguments:

- the first argument is the source image, in our case represented by the segmented ROI;
- the second is contour retrieval mode;
- the third argument is the method for approximating the contour.

The contour retrieval mode is linked to the concept of hierarchy[1]. Since knowing the hierarchy of contours is not of interest in our case, no retrieval mode was used. Finally, as already explained, the function stores the coordinates of the pixels that form the contour. However, it is not necessary to store every single coordinate in all applications: the contour approximation mode allows you to decide the degree of approximation, i.e. the number of coordinates needed to effectively represent a contour. Using cv2::CHAIN_APPROX_SIMPLE, the function will remove all redundant points that are not needed to define the contour, thus saving memory.

### 1.3.2 How to draw contours - cv2::drawContours:

The function has five arguments:

- the first argument is the source image. In this case, I used a frame specifically created to highlight only the contour of the hand;
- the second argument is the contours which should be passed as a Python list;
- the third one is the index of the list specified in the previous argument: only the outline corresponding to the specified index will be drawn. To draw all the outlines in the list, simply pass –1 as the argument;
- the fourth and the fifth arguments specify the colour and thickness of the contours. respectively.

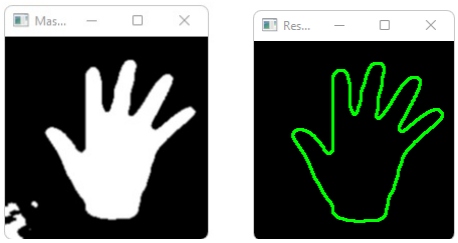
In Figure 3 an example of the result obtained.


**Figure 3:** Hand contours obtained from the segmented ROI.

### 1.3.3 Contours' optimization

In the section on segmentation [section 1.2], two problems were presented that can cause noise. Although the proposed solutions are quite effective, it is possible that, due to changes in lighting conditions or background (or in general changes related to the environment), the segmentation's result is defective. In such cases the contour detection would not only contour the hand, but also the outliers. Such a situation would lead to misinterpretation of gestures. The cv2::contourArea function was used to compute the area of the outlined regions. Considering the way the system has been set up, the area of the outliers is unlikely to be larger than the area of the hand. Therefore, the area of each contour is computed and only the contour related to the greatest one is taken into account and saved in a variable [Figure 4].

[1]the function also returns the hierarchy of the contours that divides them into levels according to the criteria of the selected mode. In general, an outer contour is considered a parent, while an inner contour is considered a child




**(A)** **(B)**

**Figure 4:** (A) Result of partially wrong segmentation, indeed we can see outliers in the lower left corner. (B) Contour of the sole hand, despite the presence of outliers in segmentation result.

## 1.4 Convex hull

The contour of the hand is then used to find the smallest convex set that contains the shape of the hand itself, namely the convex hull. The cv2::convexHull function, supplied by OpenCV, takes as input the contours and returns as output an array containing the coordinates of all the points that form the convex hull. The convex hull is then drawn via the cv2::drawContours function previously described [Figure 5]. Indeed, the function allows not only to draw the contours of an object but also any shape, provided you have its boundary-points available.

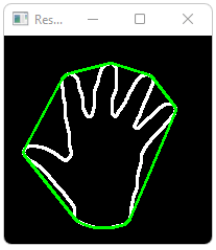



**Figure 5:** The resulting convex hull is highlighted in green.

## 1.5 Contour and convex hull properties

Some properties useful to implement the gestures are derived from contour and convex hull. Specifically, cv2::contourArea function was used to compute the area of both convex hull and contour of the hand. Afterwards, the percentage of the area not been covered by the hand in the convex hull was computed:

$$A_r = 100 \times \left( \frac{A_h - A_c}{A_c} \right) \tag{1}$$

where $A_h$ and $A_c$ are the areas covered by the convex hull and the contour respectively , while $A_r$ stands for area ratio, which is the name decided to identify this percentage.

Other properties widely used to implement gestures include extreme contour points, in particular the leftmost and the topmost points [Figure 6].

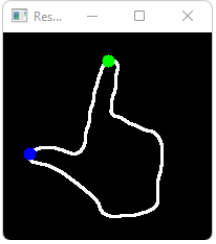



**Figure 6:** Hilighted in blue the leftmost point of contour, while in green the topmost one.

## 1.6 Convexity defects

The next step for gesture implementation is to find convexity defects, i.e. any deviation of the object – namely the hand – from the convex hull. In our case the space between two fingers is the convexity defect to be exploited.

### 1.6.1 How to find convexity defects - cv2.convexityDefects.

The contour of the hand and the convex hull are passed as arguments to the function cv2::convexyDefects, although the cv2::convexHull function needs a small change: the flag returnPoint must be set to False. Indeed, by default this flag is True, in which case the function returns all the points of the convex hull. Instead, setting returnPoint=False, it returns indices of the convex hull points. The function returns as output a matrix in which each row is represented by 4 elements: the first three elements are index referring to points contained in the hand contourn list that describes a convexity defect, we will refer to this point as start, end and far [Figure 7], while the last element is the approximation of the distance between the farthest contour point (i.e. the far point) and the hull. To identify a convexity defect we will refer to the far point: we will have as many convexity defects as the number of far points.

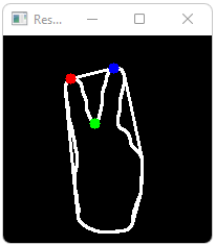


**Figure 7:** In red the end point, in green the far point and in blue the start point

The segmentation of the ROI and its contours play a fundamental role in the detection of convexity defects (i.e. far points). It is impossible for the segmentation and therefore the contours to follow the shape of the hand perfectly and, above all, to remain unchanged over time. This, as well as the shape of the hand itself, causes the presence of undesirable convexity defects [Figure 8].

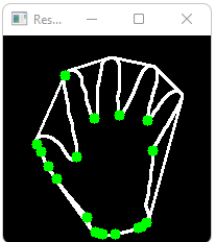


**Figure 8:** A number of undesirable convexity defects other than the space between two fingers.

For these reasons it is necessary to impose constraints to consider only convexity defects between two fingers, discarding the others. Considering that the angle between two fingers is less than 90 degree is a good way to fulfill the aim. Therefore, we will only consider a defect if this constraint is respected. To derive the angle, image a triangle between two fingers [Figure 9].

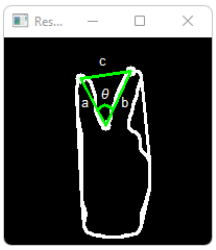



**Figure 9:** An example of how a triangle can be imaged between two fingers to derive the angle

The first step is to compute the three sides of the triangle. They were calculated using the Euclidean distance, using start, end and far points two by two. Once the three sides had been calculated, it was easy to calculate the angle using the cosine theorem:

$$c = \sqrt{a^2 + b^2 - 2ab\cos\theta} \Longrightarrow \theta = \cos^{-1}\left(\frac{a^2 + b^2 - c^2}{2ab}\right) \tag{2}$$

To highlight the defects thus obtained, a circle was drawn only on the far point using the cv2::circle function [Figure 10].


**Figure 10:** Resulting convexity defects.

## 1.7 Principal Component Analysis

The second tool used to implement gestures is hand orientation. Technically, Principal Component Analysis (PCA) was used to achieve it. This is an unsupervised learning technique for reducing dimensionality, i.e. reducing the number of variables that describe a set of data, limiting the loss of information. This is accomplished through a transformation that projects the original variables into a new Cartesian system in which the variable with the highest variance is projected onto the first axis, while the variables with the lowest variance are projected onto the subsequent axes, always following the decreasing trend of the variance. So, the co-variance matrix is used to determine the new orientation of the axes: the largest eigenvector will represent the orientation of the major axis, and so on. The function cv2::PCACompute2 was used to find only the eigenvectors, indeed the aim is to obtain the angle between them.

The first argument of the function is the dataset (contour pixels) which is re-arranged in a matrix with n rows – one for each point of the contour- and 2 columns. As a second argument it is possible to insert an empty matrix if one is interested in computing the centre of the dataset; since we are not interested in knowing it, the None value has been inserted. Then the output of the function are the eigenvectors which are used to derive the hand angle.

# 2 Gesture implementation

With this section we get to the heart of the system with a description of the implemented gestures. Tools described in section 1.6 and 1.7 have been used to carry-out gestures. According to the tool used, I classified gestures in static and non static. Basically, gestures implemented through the use of convexity defects are defined as static, while gestures implemented through PCA or mixed methods as non-static, we will see that non-static gestures are further divided into two subgroups.

## 2.1 Static gestures

Convexity defects and, to a lesser extent, the percentage of the area not covered by the hand within the convex hull (i.e. area ratio) and the area of the hand contours (i.e. area contours) were used to create these first gestures - which I call static and are the simplest in the system. Defects are simply used to detect the number of fingers shown: if defects' number is equal to one, then the number of fingers shown is two; if defects' number is equals to two, then three fingers are shown and so on, up to a maximum number of possible defects of 4, which corresponds to 5 fingers [Figure 11].

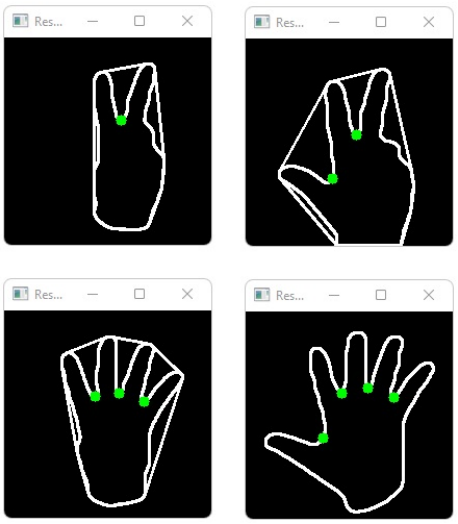
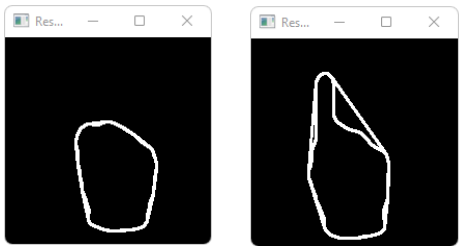


**Figure 11:** Example of how the number of defects (green dots on the images) helps to identify the number of fingers shown.

A problem arises when either none or 1 finger is shown. Indeed, the number of defects is zero in both cases. A method is therefore needed to discern the two situations. Area ratio comes to the rescue: when no fingers are shown, area ratio is almost zero; on the contrary, when the number of fingers shown is 1, the area ratio value increases [12].



**(A)** **(B)**

**Figure 12:** (A) Clenched fist: (almost) no area ratio when no finger is shown. (B) When a finger is shown, the area ratio becomes to increase.

Static gestures described so far are mainly used in the system to either confirm or invalidate the choice.

Another static gesture of the system is the palm of the hand open with the fingers stretched and close together [ Figure 13]. The number of defects is zero in this case too, so it has the same features of the clenched fist [Figure 12(A)]. A further parameter is need to distinguish the two gestures, namely the area contours: indeed, the area covered by the hand with stretched fingers is greater than the area covered by the clenched fist. Hence, setting an appropriate threshold it is possible to discriminate between the two cases.


**Figure 13:** Palm of the hand open with the fingers stretched and close together. As we can see is evident that the contour area in this case is greater that the contour area shown in Figure 12(A) when the fist is closed.

The 'stretched fingers' gesture will be used in combination with another one described below.

Since the contour area – the area covered by the hand – is essential for the proper functioning of the gesture, I decided to insert a rectangle in the ROI to guide the user to a correct hand positioning for optimal use of the system. The task was realised via the cv2::rectangle function.

## 2.2 Non stastic gestures

Non-static gestures are described below. As already pointed out,they have been further subdivided into two subgroups, which I have called dynamic and pseudo-dynamic gestures.

### 2.2.1 Pseudo-dynamic gestures.

I define the gestures described in this paragraph as pseudo-dynamic, meaning that they involve a movement of the hand albeit PCA was not needed to realize them. Two additional gestures were developed exploiting convexity defects and extreme contour points. They are widely used to edit, save and delete images. The first gesture [Figure 14] exploits the topmost point, i.e. the most high point of the contour. The topmost point is activated in the system when only a finger is shown, together with other constraints depending on the status of the system (e.g. to delete or save an image). Please refer to the following chapter for further details

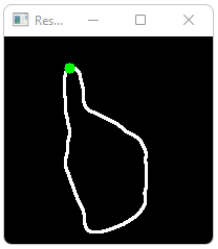



**Figure 14:** When a finger is shown and other constraints are met, the contour topmost point is activated.

The second gesture, which I will henceforth name 'pinch gesture', has been implemented in two different ways, we can decide whether to use one or the other according to a variable to set before starting the system. This gesture is activated, in certain system states, when the number of defects is equal to one (we will see later that, in order to use this gesture, it is convenient to consider a defect even if the angle between two fingers is greater than 90 degrees. In this way, the angle between index finger and thumb can be exploited). Basically, the pinch gesture allows to make change according to the

distance between two points that are attached to the tips of the thumb and forefinger. The two version of the gesture differ on how these two point are attached to the fingertips, i.e. on how the points are computed.

The first version of the gesture uses both the topmost and leftmost points [Figure 6], which are joined by a green line drawn through the cv2::line function. The green line is displayed when the gesture is used [Figure 15]. Its length, computed through the euclidean distance, varies dynamically according to the distance between topmost and leftmost.

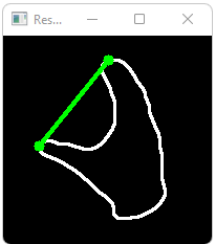



**Figure 15:** A green line join the leftmost (thumb) and topmost (index finger) points and appears every time this gesture is used.

The problem is that the leftmost point and the topmost point do not always coincide with the fingertips [Figure 16].

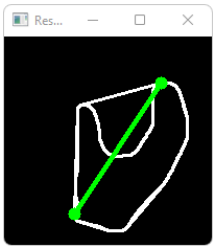



**Figure 16:** As a result of a hand rotation the leftmost point no longer coincides with the tip of the thumb.

Therefore, to obtain a line which joins the fingertip, the hand must be placed on such a way that the topmost point coincides with the tip of forefinger and the leftmost point with the tip of the thumb. For this reason a second version of the pinch gesture has been implemented.

In the second version of the pinch gesture, rather than using the leftmost and the topmost points as the extremes of the line, the start and end point, found previously by means of the function cv2::convexityDefects, are used. But also in this case there is a problem: as already pointed out, to use the pinch gesture we have to consider a defect even if the angle between two finger is greater that 90 degrees, so as a result we have more than one couple of start end points (because we have more than one convexity defects), so more than one start-end distance, i.e. more than one line as we wish [Figure 17].

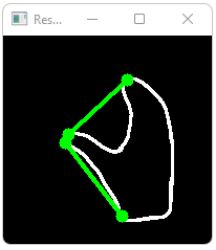



**Figure 17:** Due to the presence of more than one convexity defects we have more than one couple of start end points, so more than one line.

To solve this problem it is necessary to set an additional constraint. I notice that the line of interest has a much greater distance to the corresponding far point than the other lines, so we can use this distance as

threshold: if the distance between the given line and the corresponding far point is less than a suitable threshold, the line is not considered [Figure 18]. The threshold has been chosen empirically so as to ensure that only the start and end points corresponding to the fingertips of the index and the thumb are joined by a line, so only this distance is computed.

**(A)** **(B)**

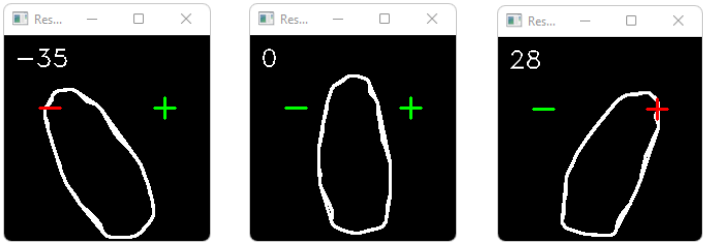
**Figure 18:** (A) A green line join the start and end points. (B) After a hand rotation, the start and and end points are still attached to the correct fingertips.

This second solution is more robust with respect to the movement of the hand, such as a rotation [Figure 18(B)], while the first one is a little bit more precise. I decided to implement both solution - because both have their pros and cons - by setting as default the second one described.

Regardless of the version adopted, from now on, we will refer to the points necessary for the construction of the pinch gesture in a more general way, i.e. as $P1$ and $P2$; while we will refer to the distance that divide them as $d$

### 2.2.2 Dynamic gestures

Hand orientation was used to build this gesture, which is used to scroll Python lists. After running PCA and obtaining eigenvectors, hand angle inclination was computed and exploited. Then a threshold $\mu$ was set to define a range of action $[-\mu, \mu]$: when the angle is greater than the upper limit, the value of the Python lists' indexes is increased, while when it is less than the lower limit, the index value is decreased [fig 19]. The gesture is used in combination with the 'stretched-fingers' one. This choice is due to the fact that the angle of inclination of the hand is less variable with regular shapes.




**(A)** **(B)** **(C)**

**Figure 19:** (A) Inclination hand angle less than the lower limit: index value in the Python list is decreased. (B) Inclination hand angle within the interval: nothing happens. (C) inclination hand angle greater than the upper limit: index value increased.

# 3 Using gestures

This chapter describes how the combination of the implemented gestures allows to navigate within the system and edit images.

## 3.1 System's structure

After providing a description of the main tools for gesture creation, there are all the elements necessary for a general description of the system and the integration of the gestures within it.

The entire environment can be divided – from an abstract point of view – into three different levels representing three different sequential stages of image selection and editing. The system has been set up in such a way that it is not possible to pass from the first to the third level and vice versa without passing through the second. The general rule imposes that it is necessary to show the open hand to pass from a low level to a high level, therefore the transition from a low level to the immediately adjacent higher level occurs when the number of defects is equal to 4; on the contrary the closed fist is used to pass from a high level to the immediately adjacent lower level. Therefore, in the latter case, the passage occurs when the number of convexity defects is equal to zero and the value of the area ratio is very low [Figure 20].

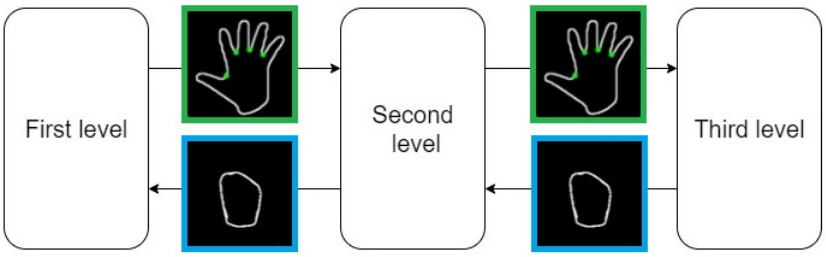



**Figure 20:** The figure shows how to move between the levels of the system.

The mode of switching between level just described has a vulnerability. Indeed, using the open hand at the first level should lead to the second level. However, once we get to the second level the number of defects detected will always be 4, which will lead directly to the third level. The same vulnerability occurs when we are on the third level and we want to go back using our closed fist: the gesture will also be detected on the second level, bringing the system erroneously to the first level [Figure 21].

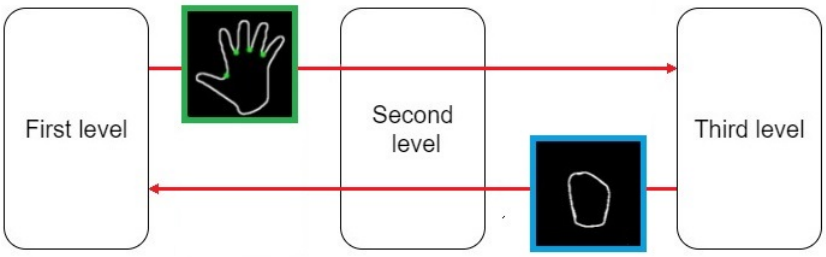



**Figure 21:** Incorrect level selection: no access to the second level.

A flag was used to solve these problems [Figure 22]. The system must not only detect the gesture but also check the status of the associated flag in order to pass from one level to the next: if flag = True, then it is possible to move to the successive level. Once the switch has been made, the flag becomes False, preventing detection of the gesture and therefore the switch to another level. To reactivate the flag and

be able to use the gestures again, it is necessary to show the system the 'stretched-fingers' gesture. Then, when the number of defects is zero, the ratio area value is close to zero and the contour area exceeds a certain threshold, the flag reactivates and in turn reactivates the gestures.

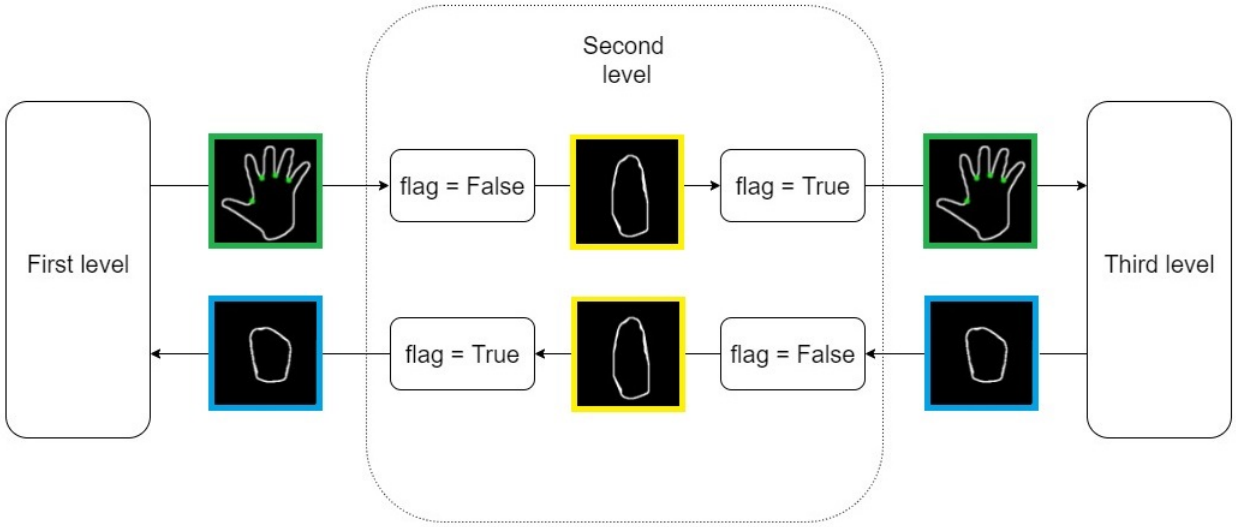



**Figure 22:** The flag allows correct selection of levels..

## 3.2 The levels of the system

We have just described how to get around in the system. Let's now move on to a more detailed description of each level.

### 3.2.1 The first level

The main functionality of the first level is the selection of the image that the user wishes to edit. The choice can be made among a set of images contained in the folder 'test images'[2] and loaded in a list via the cv2::imread function, while the cv2::imshow function allows to visualize the current image, which corresponds to an initial index of zero. Moreover, the name of the displayed image is noted inside the caption.

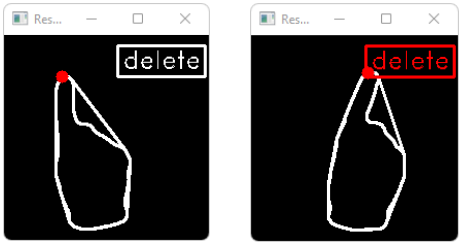
To browse the images, the angle of inclination of the hand was used, which is only read by the system when the 'scretched-fingers' palm is shown. Indeed, as mentioned above, the angle reading is more accurate under this condition. Thus, tilting the palm to the right increases the index of the list containing the images by one and the image corresponding to the updated index is displayed. However, if the index exceeds the size of the list, it is set to 0 instead of incrementing it. Similarly, tilting the palm of the hand to the left decreases the index of the list containing the images. If the index is zero, scrolling to the left does not decrease it but causes its value to be equal to the maximum value of the list. Therefore, the index of the list containg the images is updates if the grade of the palm exceeds a certain (predefined) threshold.

However, once the threshold is exceeded, the index continues to update until the hand tilt returns below the threshold value. This results in an index increment/decrement greater than one, making it difficult to select the desired image. To overcome this problem, which is very similar to the problem encountered when selecting the level, a flag has been used: the image list index will only be updated if the slope value is greater than the threshold value and the flag is True. Once the threshold value is exceeded, the index is updated and the flag becomes False. To update the index again, the flag must again become True, which happens when the hand tilt returns to a value below the threshold.

[2]Images must be placed in the folder before starting the system.

In addition to the possibility of browsing through the various images present, two further actions can be carried out on the first level, namely deleting and/or selecting an image.

Convexity defects and some properties of the contours are used to implement deletion of the current image. When the number of defects is 0 and the area ratio is greater than a certain threshold level (i.e. when only one finger is shown), the topmost point of the contour is calculated (highlighted using the cv2::circle function) and, at the same time, a box with the word "delete" appears [Figure 17(A)]: when the topmost point - corresponding to the tip of the index finger - belongs to the range of points forming the "delete" box, the current image is deleted. In simple terms, deletion takes place when the tip of the index finger goes on the "delete" box [Figure 23(B)].




**(A)** **(B)**

**Figure 23:** (A) the gesture that enables deletion of the image. (B) The actual deletion of the image.

Again, it was necessary to use a flag to ensure that deletion only affects the current image: once the image has been deleted, the flag becomes False. To reactive it – i.e. to have again the chance to delete the image – simply show the "stretched-fingers" hand.

Image selection is instead achieved by showing the open hand, i.e. 4 convexity defects. This action does nothing more than create a working copy of the designated image and change the state of the system: indeed, after selecting the image, we move on to the second level.

### 3.2.2 The second level

The second level displays only the working copy of the image selected at the first level. Conceptually, the operation of this level is similar to the previous one, but instead of scrolling through a list of images, you can scroll through a list containing the names of the operators used to modify the selected image.

The caption will show the name of the selected image and the name of the current operator, which varies according to the index value of the list. To scroll through the indices of this list, the same gesture used to browse the images in the first level is used.

Images can be saved at this level, but this action can only be performed if the working copy is different from the original image. Therefore, saving can only be performed after one or more changes have been made to the working copy. The save operation is identical to the delete operation seen on the first level, but instead of displaying the box marked "delete", it will be displayed with the word "save". By hovering the index finger over the word "save", the image will be saved in the test images folder using the cv2::imwrite function [Figure 24]. This action will bring the system back to the first level where the modified image will be displayed.


**(A)** **(B)**

**Figure 24:** (A) Saving action is activated. (B) How to actually save the image.

If the image has been selected incorrectly, it is possible to return to the first level by using the closed fist gesture, while the selection of the operator you wish to use on the working copy of the selected image is instead achieved through the use of the open hand gesture. This action will take the system to the third level.

### 3.2.3 The third level

On the third level, the actual editing of the image takes place according to the operator selected on the second level. The names of the selected image and operator will then be shown in the caption.

The operating principle of the various operators is almost the same and follows the general rule explained below. Only one operator will require an additional step.

Once the operator has been selected, it is possible to modify the image through the pinch gesture, previously described. The values of the parameters used in the various functions to modify the image will vary depending on the length of the line joining the $P1$ and $P2$ points. Saturation values, depending on the type of operator chosen, are also provided so that the parameter values do not exceed certain limits [Figure 25].

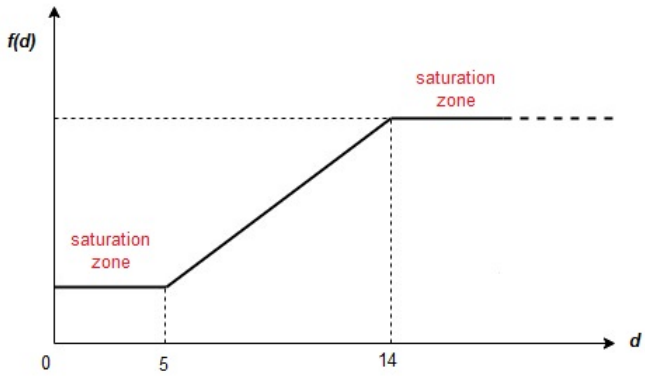



**Figure 25:** If the value of the distance $d$ lies between a range of natural values from 5 to 14 we obtain as output a function $f(d)$, which depends not only on the distance but also on the type of operator used. Basically is a mapping between the distances and the values of a certain parameter of the given operator. If the distance is outside this range, the function assumes its minimum value if the distance is less than 5, and its maximum value if the distance is greater than 14.

To understand whether these limits have been reached, a sort of battery has been inserted, created using the function cv2.rectangle: the level of filling of this battery depends on the length of the line joining the points $P1$ and $P2$. The battery will be empty when the lower limit is reached, while it will have the maximum charge when the upper limit is reached [Figure 26] The displayed image will vary dynamically depending on the values of the parameters managed by the gesture to see the effects of the change in real-time.

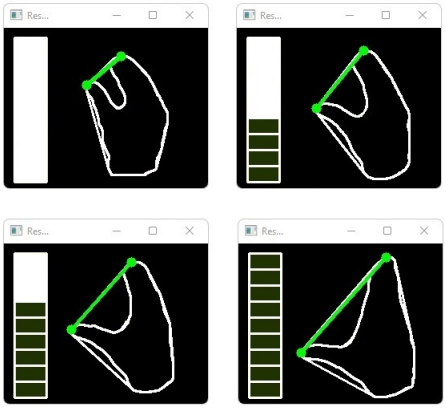
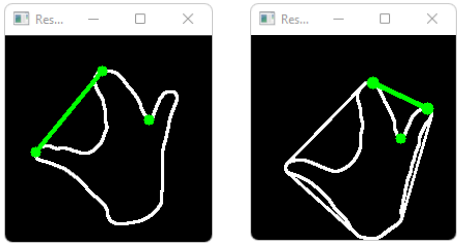


**Figure 26:** The pinch gesture used to modify images. The distance between the $P1$ and $P2$ points indicates the degree/intensity of the chosen change operator. The battery level helps to understand the minimum and maximum limits of operator saturation.

To confirm the change it is necessary to raise the little finger so that the system detects a number of convexity defects equal to two [3] However, with the 'pinch gesture' the angle between the thumb and index finger could exceed 90 degrees, in which case the defect would not be considered and, consequently, raising the little finger would lead to a defect number of 1 and the change would not be confirmed [Figure 27].



**(A)** **(B)**

**Figure 27:** In both the cases the angle between the forefinger and the thumb are greater than 90 degrees, so the number of convexity defects, highlighted in green, is only one. In the figures the results obtained when using the first version (A) and the second version (B) of the 'pinch gesture'.

Therefore, as mentioned in the section 2.2.1, only in this case the convexity defect is considered even if the angle is greater than 90 degrees [Figure 28].

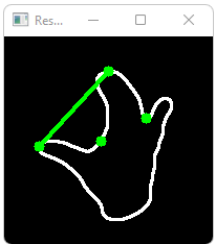
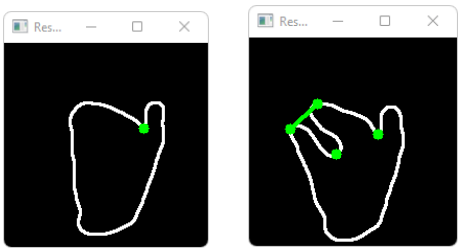


**Figure 28:** Considering a defect even when the angle is greater than 90 degrees allow to detect two defects and, therefore, confirm the change.

For the same reason, the lower limit for all operators has been set in such a way that the minimum value is obtained by holding a certain distance between the thumb and the forefinger, because if this two finger

[3] If the first version of the 'pinch gesture' is used, care must be taken to ensure that the little finger does not go over topmost, otherwise the distance would vary accordingly, making it unable to achieve the desired change value. With the second version of the pinch gesture, there is no need to take any precautions at all, which is another advantage that led me to set it as the default in the system [section 2.2.1].

were touching, the number of defects would be 0, making any attempt at confirmation by raising the little finger pointless [Figure 29].



**(A)** **(B)**

**Figure 29:** (A) When thumb and index finger touch, number of defect equals zero and no change is confirmed. (B) The right way to make the change effective, indeed, thanks to the saturation zone, bringing the two fingers even closer together would not produce any change.

Thus, when the number of defects is two the change made is confirmed and you return to the second level where you can select another operator, save the image or cancel everything.

## 3.3 Operators

In this section a detailed description of the implemented operators is provided together with the reasons that led to their choice.

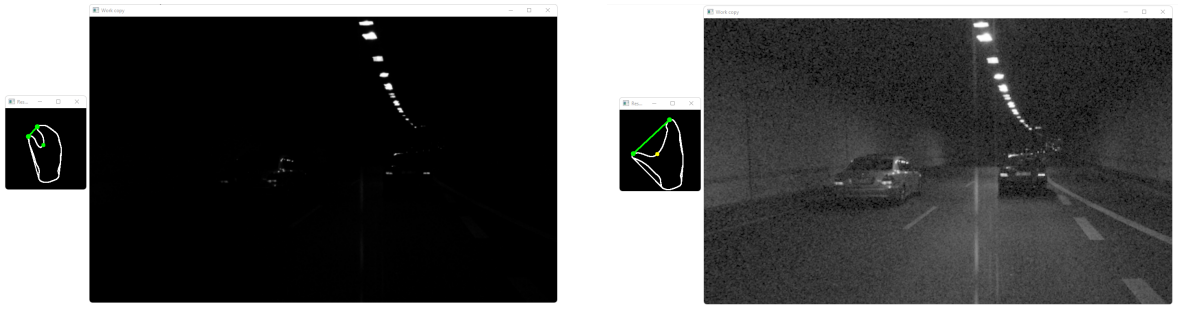
### 3.3.1 Exponential operator

To change the contrast of the images I decided to implement the exponential operator. The decision fell on this operator because it is more effective than the linear operator and more interactive than histogram equalisation. Indeed, while for histogram equalisation the intensity of the change does not depend on any variable parameter, for the exponential operator (but also for the linear operator) the intensity of the change depends on a value $r$:

$$y = \left(\frac{x}{255}\right)^r 255 \tag{3}$$

The pinch gesture acts on the $r$ value. Its minimum value has been set to 0.4, its maximum to 1.75, with an interval between values of 0.15.

An example of the use of the exponential operator is shown in Figure 30 below.



**(A)** **(B)**

**Figure 30:** (A) The distance $d$ between $P1$ and $P2$ is small, as a result the image tends to become darker. (B) The distance $d$ is large, as a result the image tends to become brighter.

### 3.3.2 Spatial filters

The steady Gaussian filter was implemented among the linear spatial filters, using cv2::GaussianBlur function. The function has 4 arguments:

1 source image

2 Kernel dimension

3 Standard deviation in x direction

4 Standard deviation in y direction

However, the pinch gesture only affects one of the parameters, i.e. the kernel, while the standard deviations have been set to a fixed value of 0, which cannot be changed by the system's built-in gestures. This is because when the value of the standard deviations is 0, they are calculated directly by the function based on the size of the kernel. The kernel must have an odd size: all odd values between the minimum size - set at 1x1 (no effect) - and the maximum size - set at 19x19 - have been taken into account.

An example of the use of the Gaussian filter is shown in Figure 31.

**(A)** **(B)**

**Figure 31:** (A) The distance $d$ is small, so the size of the kernel and the level of smoothing are low. (B) The distance $d$ is large, thus the kernel is also large and the level of smoothing is high. Large kernels tend to blur edges.
Note that when the distance between the fingers is less than 90 the far point is highlighted in green, while if the distance is greater than 90 the far point is highlighted in yellow

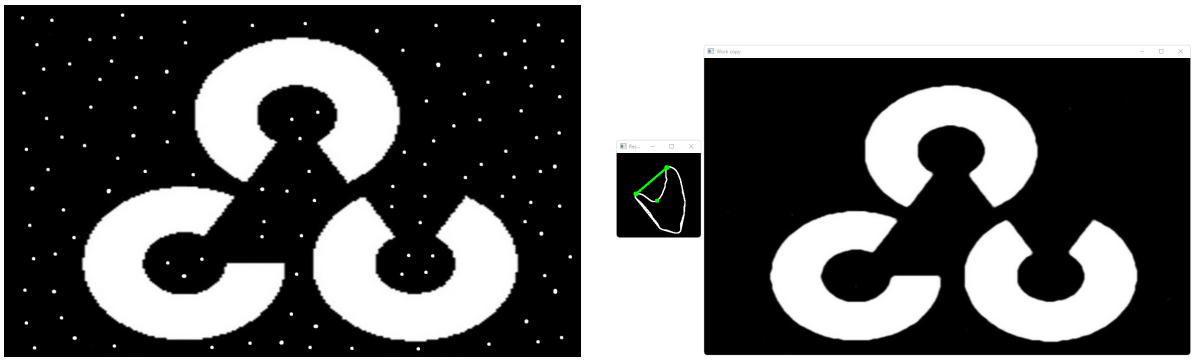
The Gaussian filter is intended to reduce the noise, but has the side effect of blurring the image [Figure 31(B)]. For this reason, a further filter was implemented with the same purpose but without the side effect of blurring: the bilateral filter [Figure 32] . To achieve filtering, the function cv2::bilateralFIlter is used, which has: source image, kernel dimension, two sigma values. For simplicity, the latter were set to the same value. If their value is less than 10, the filter has little effect; if their value is greater than 150, the filter has a very strong effect. That's why an intermediate value of 75 was chosen. So here too, the pinch gesture acts on the size of the kernel in much the same way as the Gaussian filter.

**(A)** **(B)**

**Figure 32:** (A) Image affected by Gaussian noise. (B) Thanks to the bilaterl filter, Gaussian noise is reduced without blurring the edges.

The bilateral filter is a non-linear filter that is much slower than the Gaussian filter and therefore not very suitable for real-time applications. However, it is very effective in performing noise reduction while keeping the edges sharp enough, which is why it has been implemented anyway despite its slowness.

The local filters described so far are mostly used for Gaussian noise; however, they are not very suitable for dealing with impulse noise. For this reason I decided to implement an additional non-linear filter specifically to handle this kind of noise. The filter in question is the median filter and has been implemented using the function cv2::medianBlur. There are only two arguments for this function, i.e. the source image and the size of the kernel; so, also in this case, the pinch gesture acts on the size of the kernel [Figure 33].

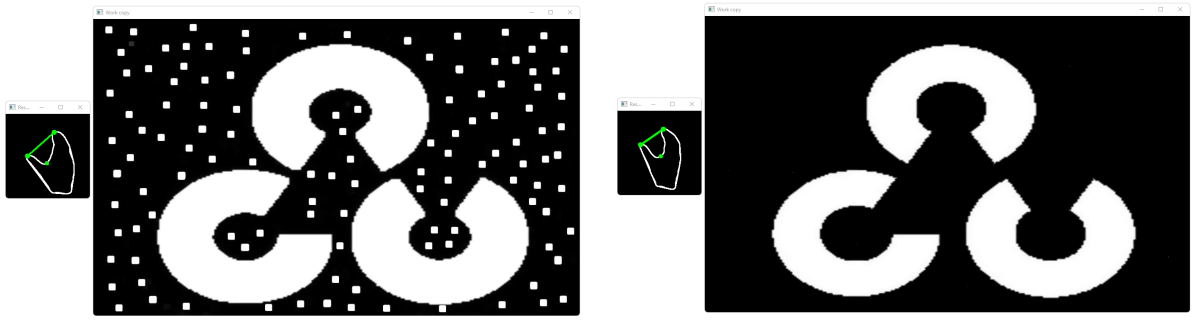


**(A)** **(B)**

**Figure 33:** (A) Image showing some outliers. (B) The result after applying the median filter.

### 3.3.3 Morphological operators

Morphological operators implemented in the system are dilation [Figure 34(A)] and erosion [Figure 34(B)]. Functions cv2::dilate and cv2::erode were used. Both have the same arguments: source image, kernel and number of iterations, i.e. how many times function must be applied. Once again, pinch gesture acts on the size of the kernel. In morphological operators, kernel can have any shape depending on its use. For simplicity's sake, a classic $n \times n$ kernel was used – with n odd – suitable for all situations. The number of iterations was set to 1.



**(A)** **(B)**

**Figure 34:** (A) Dilation applied to Figure 33(A). (B) Erosion applied to Figure 33(A).

Dilation and erosion are the basic morphological operations. The decision to implement them is based on the fact that they allow to carry out opening and closing operations without the need to implement further functions. Indeed, closing is nothing more than dilation followed by erosion, while opening coincides with erosion followed by dilation.

### 3.3.4 Crop

The only operator whose functioning is slightly different from the others is the image cropping. It is divided into two stages. The first one is very similar to the one of other operators: the pinch gesture

enables to manage the size of a square placed in the centre of the image [Figure 35] [4]and, as for the previously described operators, raising the little finger (thus allowing the system to detect 2 defects) confirms the size of the square. Nevertheless, in this case confirming the size of the square does not coincide with confirming the change, as this action will not lead to the second level of the system but to the second stage of the operation, i.e. moving the square within the image to decide which portion of the image to crop [Figure 36].

**The first stage** The square on the working copy of the image is created using the function cv2::rectangle and, again, its size depends on the length of the line joining the $P1$ and $P2$ points of the pinch gesture. The side of the square has been designed to not exceed the length of the smallest dimension of the selected image. In this way, due to its central position, it is not possible for the square to exceed the size of the image in the first phase.

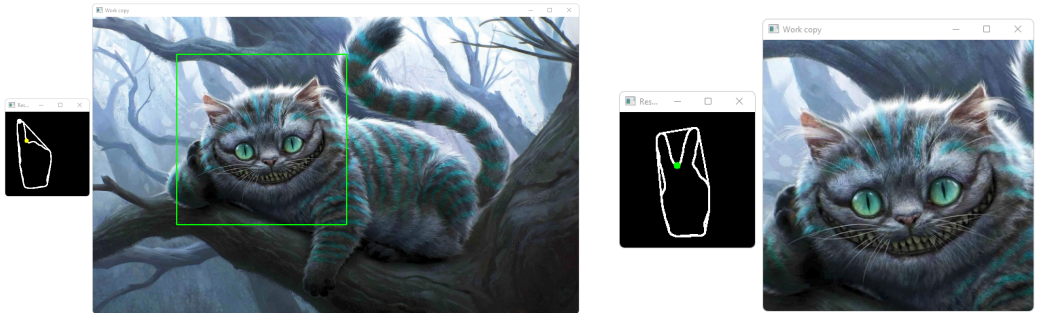
**Figure 35:** Handling the size of the square drawn on the image.

**The second stage** During the second stage, the square is dragged within the image, following the movement of index finger's tip. Once again, the topmost point is exploited. Topmost point is actived when the number of defects is 0 and ratio area has a positive value far from zero. Therefore, the staring point of the square (the top left vertex) will depend on the position of the topmost point; while the ending point (the bottom right vertex) will depend on the starting point and on the length of the side set during the first stage. While the dimensions of the images the user wants to modify can vary, the dimension of the ROI where to place the hand remains unchanged. Moreover, the dimension of the ROI will almost certainly be smaller the image's dimension: it will have fewer pixels. For these reasons, in order to allow the square to be dragged within the entire image, regardless of both the size of the square and the size of the image, a normalisation operation is performed between the dimensions of the ROI and the dimensions of the selected image. Moving the square after normalisation will be less fluent, but it will reach all the points of the image. Again, saturation conditions have been included so that the square does not exceed the size of the image.

**Figure 36:** Handling the position of the square drawn on the image.

[4]The square is used to select the area of the image the user wants to keep when cropping the image.

Showing two fingers allows to confirm the position of the square [Figure 37]. Once the position of the square has been confirmed, the system will a slice operation that will depend on the start and end point of the square. In addition, an offset equal to the thickness of the square line has been added to the slice to prevent it from being displayed in the final image.



**(A)** **(B)**

**Figure 37:** (A) Positioning the square . (B) Slicing of the image after showing two fingers.

Once the slice operation has been carried out, the system will return to the second level where the user can make other changes, save the image or cancel all changes made so far and return to the first level.

# Conclusion

To conclude, the system responds rather well to the implemented gestures. They allow to modify images without noteworthy problems (even if some practice is required for an optimal use). The proper functioning of gestures depends particularly on the success of segmentation; therefore, the environmental conditions in which the system is executed are fundamental.

Finally, I would like to point out that all the results obtained are shown directly on the frames captured by the webcam, not on a black frame as shown in the report. The choice to show the results obtained on a black frame was simply to highlight the results, but in use it is much more intuitive to show them directly on the frame for an optimal use of the system.