INF573

Dynamic Mouse

Computer vision

Hichem BOUCHAIB

Environment :

OS : macOS Mojave (10.4.6)
IDE : xCode (11.2.1) with OpenCV 3.0

Objectives:

The objective of this project is clear: you have to be able to move the mouse cursor... without touching the mouse!
In other words, it is necessary to make the mouse dynamic while keeping a distance with the computer.

To do this, we are going to move the cursor with our hand. Our program must be able to recognize and follow the hand. Around our hand, the program will draw a rectangle whose center corresponds to the position of the dynamic cursor.
Then we should be able to generate a left mouse click by color recognition.
Indeed, when the user wishes to make a left click, he must turn his hand over to reveal a blue color patch. Once the color has been detected, the system translates this into a left click.

You can see the demo here :
Hand Tracking : https://youtu.be/-0gHWn1T9rI
Mouse Moving : https://youtu.be/bNElZO8QxvI
Mouse Clicking : https://youtu.be/5xloW0aQScs

Here is the **Github** : https://github.com/Hichem-Bouchaib/INF573_Project

Note : If you are on macOS and want to execute the code. The program will take control of your mouse. If you do not want to, you can uncomment the line 75 of handContour.cpp that call the mouse functions.
You will also need to specify the path to the .xml face classifier file that is stored among the .hpp et .cpp files. You can do this in the line **46 on main.cpp**

I- Detect and remove the face

As we will see later, our system is based, among other things, on skin detection.
It's great for recognizing the hand, but it becomes quite complex when you add your face to it. To make these parts more independent from each other, I decided to follow PierfrancescoSoffritti's advice by "removing" my face from the frame.
Specifically, I will put a black rectangle that will cover my entire face and neck, in order to interfere as little as possible with the detection of the skin on my hand.
In my code, this is ensured by the function `void detectAndDisplay( Mat frame )`.
In this function, I'm using an existing frontal face classifier *"haarcascade_frontalface_alt.xml"*.

In this function, I use an already existing frontal face classifier "haarcascade_frontalface_alt.xml" which makes my work easier. Then, a simple call to the function `face_cascade.detectMultiScale( frame_gray, faces );` will allow me to detect the faces.
But that's not enough. To make it efficient, I still have to cover my face with a black rectangle. This is what the end of the *detectAndDisplay* function does. I simply retrieve the center of the face and then determine the size of the rectangle to cover the entire skin surface.
For more information, please have a look at the *detectAndDisplay* function, directly accessible on the *main.cpp* page.

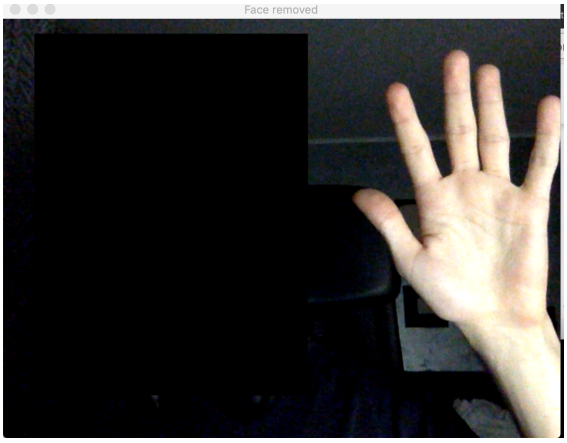


Figure 1 result from the face removal

II- Background Removal

After deleting the user's face, it's time for us to do some image processing.
One of the great difficulties when you want to track an object is to ignore the background, which is supposed to remain static.
Although I'm doing skin detection here, I felt that removing the background upstream was more sensible at best, especially to limit noise.
We will see later, but skin detection is not perfect, and depends mainly on light. To limit this problem as much as possible, removing the background may be a good idea.

But before proceeding with image processing, it is obvious that it is better to first **blur the image to reduce detail and noise**. This is what I do thanks to the Gaussian blur function.

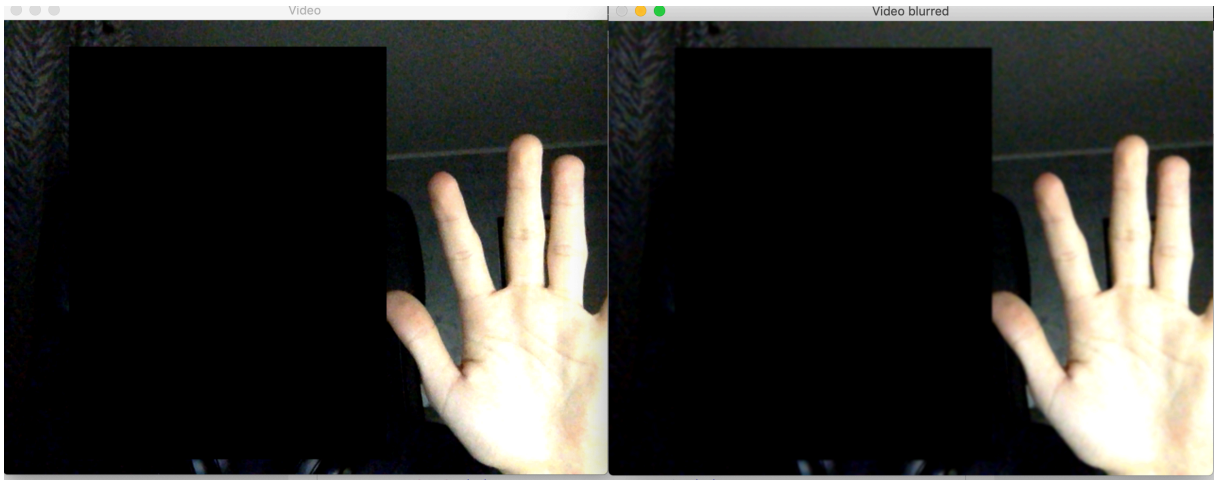


Figure 2 not blurred - blurred video

Once that's done, we'll be able to proceed with the background removal.
There are several techniques for doing some background substraction. Personally, I have chosen to go for BackgroundSubtractorMOG2. It is a Gaussian Mixture-based Background/Foreground Segmentation Algorithm. As Zakaria said on StackOverFlow, *"one important feature of this algorithm is that it selects the appropriate number of gaussian distribution for each pixel."*
Therefore, it provides better adaptability to varying scenes due illumination changes etc.

The ouput gives us a mask where pixels in white are considered to be part of the foreground, and the black part of the background. We can notice that there are still some noises, but at this point it is not really important. As long as we can easily recognize the hand from the background, it makes it suitable.

That's cool, but do not forget that we will have to compute skin detection. At this point we have a binary image, so it absolutely not compatible with what we want to do. We need to go further on our reasoning!

In the 2 lines below, we are creating a new matrix with same size than the frame we got from our webcam. This matric is full of zero, so it is black.

Thanks to the MOG2 background subtraction, we got a mask. A mask is a binary image of the same size as image we are working with. The function will work only with pixels that are not zero in the mask. Therefore, it would be interesting to copy a new frame from the one we got from the webcam, and then apply the mask from MOG on it. That's what we are doing here !

```
cv::Mat colorForeground = cv::Mat::zeros(cameraFeed.size(), cameraFeed.type());
cameraFeed.copyTo(colorForeground,fgMaskMOG2);
```

The result is pretty much cool. We have or background colored in black, and the foreground colored by its natural color(s).

Figure 3 Foreground

III- Skin detection color

The skin detection is fully implemented in the *skinColor.hpp*.
The function is taking as input the foreground we previously processed. After that, the function will convert this function to HSV. In several forums, I read that HSV is better for object detection. More precisely, because the R, G, and B components of an object's color in a digital image are all correlated with the amount of light hitting the object, and therefore with each other, image descriptions in terms of those components make object discrimination difficult. Descriptions in terms of hue/lightness/chroma or hue/lightness/saturation are often more relevant.
HSV color space is consists of 3 matrices, **'hue'**, **'saturation'** and **'value'**. In OpenCV, value range for 'hue', 'saturation' and 'value' are respectively 0-179, 0-255 and 0-255. 'Hue' represents the color, 'saturation' represents the amount to which that respective color is mixed with white and 'value' represents the amount to which that respective color is mixed with black.

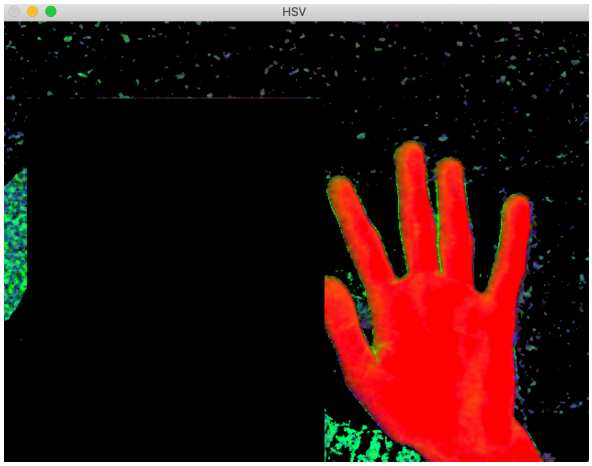
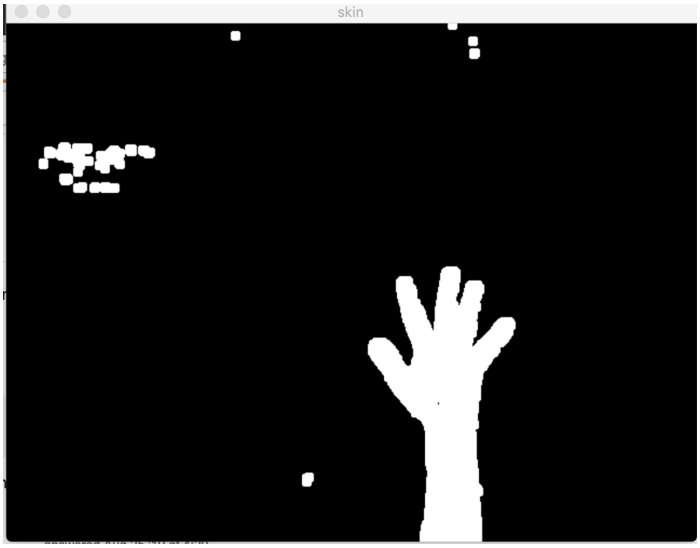


Figure 4 frame converted to HSV

Then, I need to define a range (low to max) for the skin's color. For that, I followed the advice found on StackOverFlow, and I defined these values:
Low = Scalar (0, 48, 80)
Max = Scalar (20, 255, 244).

It works pretty well for what we want to do, but we should keep in mind that it also depends too much on light condition.



At this point, the **hand is easily recognizable.** We still have some noises but finding the biggest contour will solve the problem. We'll talk about it later on.

IV- Hand contour

Now, it's time to draw the contour of our hand, and only this one.
That what we will do in the handContour.hpp.

Our function will take as input the image above and will output an image with the hand contour drew in green.
Basically, we are using the function `findContours(input_image, contours, hierarchy, RETR_EXTERNAL, CHAIN_APPROX_NONE);` to identify all the contours of our image.
Contours is a type of array that stores all the contours. As we have processed the image previously, our hand is supposed to be (most of the time) the biggest contour.
Then by doing some math, we can easily determine the contour that has the biggest area. This will be our hand, and we shall draw the contour of only this shape.

Because we need to project the position of our hand in a 2-D dimension, it is important to compute its position. In order to do that, the function *convexHull* is very useful.

```
convexHull(Mat(contours[biggest_contour_index]), hull_points, true);
```

It will draw the convex hull of our hand. This shape will allow us to do some math (once again), to determine the center of it. Its position will be the position of our cursor.
All the math is explained is the function *handContour()* that you can find on.
Then we just draw the biggest contour in green, the rectangle (convexhull) in red, and the center point with its coordinates.

```
drawContours(contours_image, contours, biggest_contour_index, Scalar(0, 255, 0), 2, 8, hierarchy);
rectangle(contours_image, bounding_rectangle.tl(), bounding_rectangle.br(), Scalar(0, 0, 255), 2, 8, 0);
circle(contours_image, center_bounding_rect, 5, Scalar(0, 0, 255), 2, 8);
```

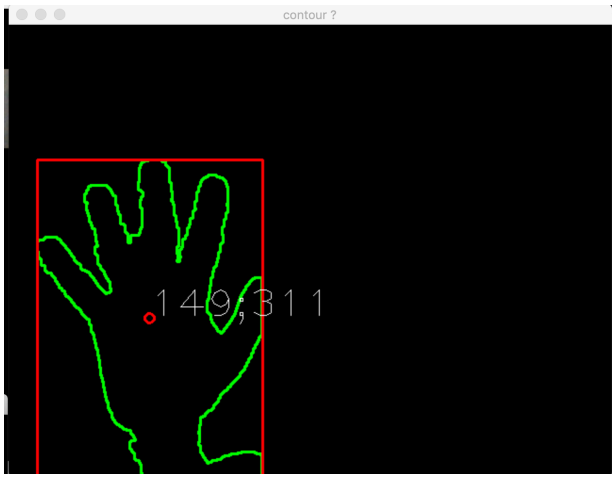


Figure 5 Our hand with its contour, convex hull and center point

V- Move the mouse

Ok so now we are able to track the hand and compute its center position. It is time to move our cursor depending on its position (*mouse.cpp*)
As I am using MAC operating system, I have to deal with the CoreGraphics framework to get access to my computer mouse. The docs are pretty well informed, and it was quite easy to implement https://developer.apple.com/documentation/coregraphics (that said, accessing the webcam through xCode was a nightmare).

As I am moving my hand, the function handContour will still compute the new position. This position will be sent as parameter to 2 functions:

1)

```
void moveMouse(float x, float y){
    CGDisplayMoveCursorToPoint (kCGDirectMainDisplay, CGPointMake(x, y));
}
```

moveMouse will order the cursor to move according to CGPointMake, the point computed with our hand's center coordinates

2)

```
void clickMouse(float x, float y){
    CGEventRef click1_down = CGEventCreateMouseEvent(NULL, kCGEventLeftMouseDown, CGPointMake(x,y),
        kCGMouseButtonLeft);
    CGEventRef click1_up = CGEventCreateMouseEvent(NULL, kCGEventLeftMouseDown, CGPointMake(x,y),
        kCGMouseButtonLeft);
    CGEventPost(kCGHIDEventTap, click1_down);
    CGEventPost(kCGHIDEventTap, click1_up);
}
```

clickMouse that will simulate a left click of our mouse. First of all, we click down, and then click up. As we are doing with a natural mouse.

Notes: I have troubles to make it work. The first time I computed my program it asked to accept the right to access to my mouse. I accidentally refused, and I did not find a way to fix this problem (even in the Preferences System).
Nevertheless, I implemented it, and instead of clicking, I am printing "clicked" as output of our program to demonstrate that the click works when it should work.
By the way, how shall we tell the program to click? Answer: When the user flips its hand to reveal a blue dot !
So… it is time for the **color detection**!

VI- Color detection

The principle is very simple. First, I define a color range of blue to be detected:

```
inRange(color, Scalar(110,50,50), Scalar(130,255,255), mask1);
```

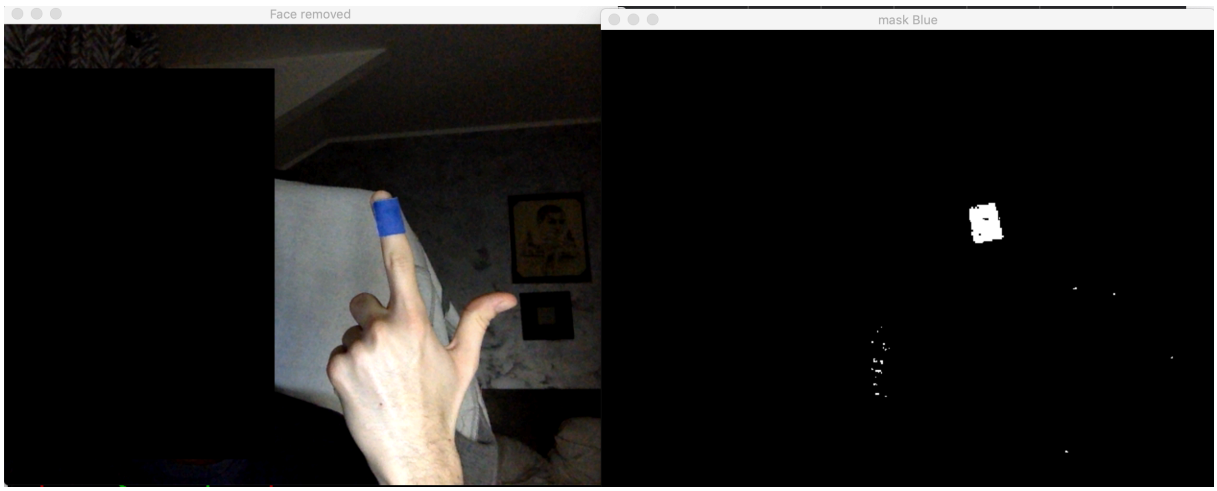
Then I convert once again the frame from the camera to HSV. This output will be processed by *inRange()* that will catch the colors in this range of values to produce a threshold. Its output is mask1.
Below, you can see the frame from the camera (left), and the mask processed (right).

Then it is easy. As we have done previously, we just need to take into account the biggest contour (to avoid noise). Then to determine if it is a click, I am saying something like that *"Well, the blue is detected. Is it really the blue dot ? If yes, well let's go for the click. If it is the noise, then we do not click".*

Thanks to the function

```
if (countNonZero(input_image)>800){
    clickMouse(center_x, center_y);
    cout<<"Click detected"<<endl;
}
```

I am determining if the contour has at least 800 pixels in white. If yes, it means that it is our blue dot, and so we click. If not, well… we don't click!
Of course, this part is upgradeable, but it is enough for what we want to do so far.



Conclusion :

In my situation it works pretty well. A little bit sad that the left click did not work in the real world, but it is was a good training to understand and implement all the different concepts.
What I faced is a huge problem on computer vision: it depends on so many parameters that it is quite hard to make it scalable.
Basically, it's hard to have one fixed color range for skin, because even if you want to detect only your own skin, its color will actually change a lot depending on lighting conditions.
It's the same for the blue color detection.

What I was thinking about to improve my program: As we can detect the face easily, we could compute the average of the face color automatically, and then use the values computed in our HSV.
Or we can do put our hand in a square, and calculate the average of the skin color, and use it afterwards, as I have seen some codes about this approach. However, I wanted to get something automatic, that do not impose to the user to do anything to make it work.
We can also make the blue color detection more robust like storing the values in a FIFO queue and determining the average to take a decision.

Sources :

https://answers.opencv.org/
https://computergraphics.stackexchange.com/
https://stackoverflow.com/
PierfrancescoSoffritti
https://developer.apple.com/documentation/coregraphics
https://docs.opencv.org/