Open CV Project

Group 1

RAMÓN GONZÁLEZ, Miriam - 60577

RAMIRO BONILLA, Celia - 60576

PHAN, Antoine - 9462

BIENFAIT, Nathan – 9361

DIEZ, Theophile - 10769

Contents

1. Tasks

Functions	Student
Dilatation/Erosion	Nathan BIENFAIT
Resizing	Miriam RAMÓN GONZÁLEZ
Lighten/Darken	Antoine PHAN and Celia RAMIRO BONILLA
Panorama/Stitching	Theophile DIEZ
Canny Edge detection	Celia RAMIRO BONILLA

Additional Functions	Student
Crop	Miriam RAMÓN GONZÁLEZ
ColorMap	Celia RAMIRO BONILLA

2. Functions

Dilatation/Erosion

For these functionalities, it has been created four functions. Two for dilatation and two for Erosion. The functionalities are implemented the same way, but we choose to separate them, so we can choose to dilate or erode an image.

1. `Mat Erosion (cv::Mat img); & Mat Dilatation (cv::Mat img);`

   Those two functions work the same way. First, they make sure the Mat dst, which is where we will store the output image, object is empty. Then they create a window and two trackbars. The first trackbar is to choose the shape of dilatation/erosion, it can be a rectangle, a cross, or an ellipse. The second trackbar is to control how much we dilate/erode the image. Those last trackbars call respectively the functions that we will explain later. When the user has finished by pressing a key, the functions clear the shape selected, clear the amount of dilatation/erosion applied and return the image created as an output.

2. `void Ero (int, void*); & void Dil (int, void*);`

   Those are the functions respectively called by the trackbars. First they both call the OpenCV function Mat getStructuringElement(int shape, Size ksize, Point anchor=Point(-1,-1)) which allow to specify the shape selected, the size of kernel to dilatation/erosion and the point where to apply it. These function returns a kernel that we are going to use in the OpenCV dilate(src,dst,element)/erode(src,dst,element) functions. Those two last functions take the source image, apply the kernel on it so that bright/dark areas get bigger and store the output in the output image.

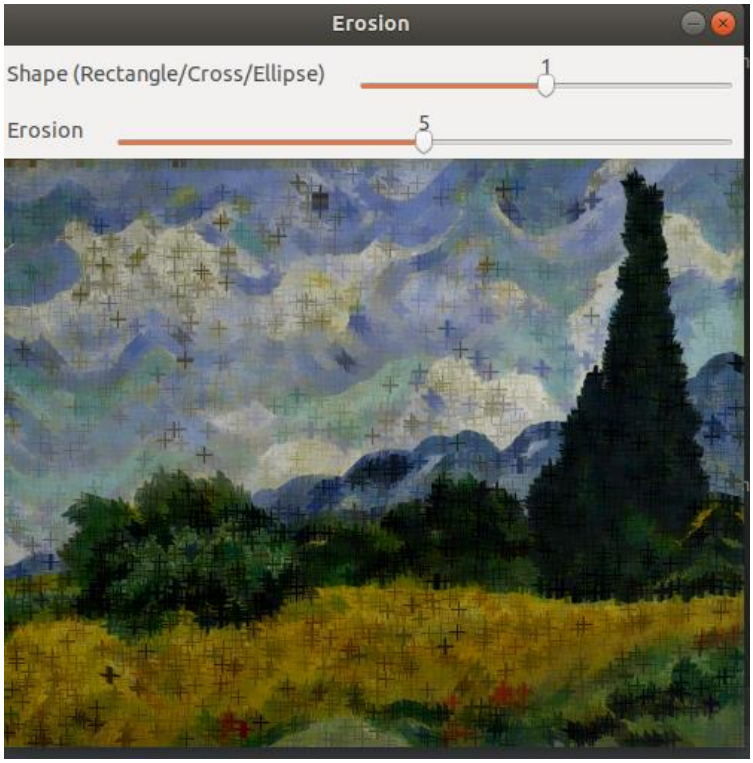


Figure 1. Erosion with cross shape

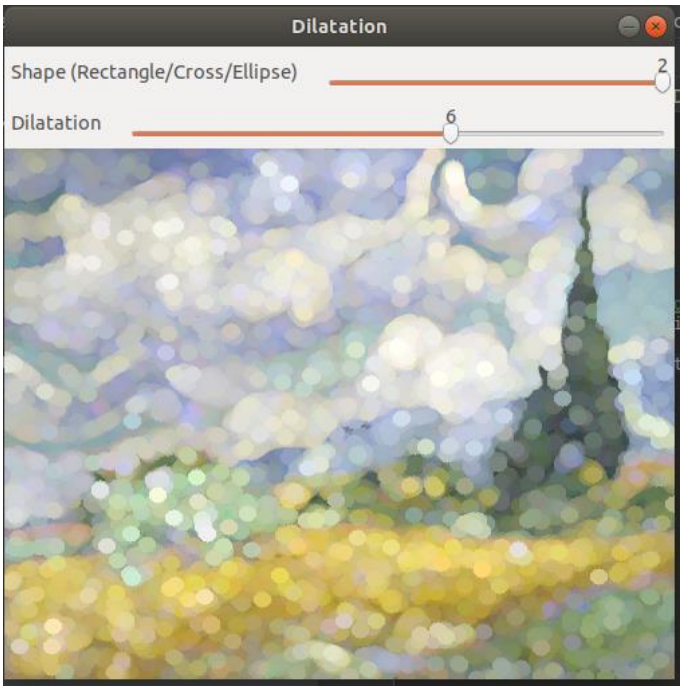


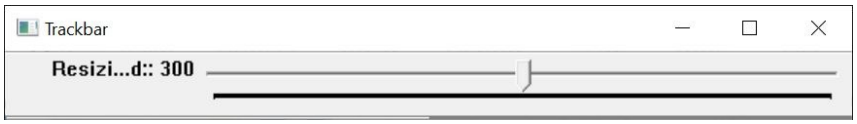
Figure 2. Dilatation with ellipse shape

Resizing

For this functionality is been created two functions:

1- `void Resize (cv::Mat img);`

a. Store in a local variable the image that is passed as an argument.
b. Create a window where the trackbar will be displayed.
c. Create a trackbar to manage the factor of resizing.
   - The factor has a value between 0 and 600.
   - The trackbar starts in the mean value (300).
   - Initially two trackbars were created to manage the factor resizing of width and height. It works properly but it doesn't keep the aspect ratio of the image.
     Finally, only it is used a single trackbar to manage proportionally width and height in order to avoid image distortion.



d. Call to resizing function.

2- `void resizing (int, void*);`

a. Call to OpenCV resize function → `void resize (InputArray src, OutputArray dst, Size dsize, double fx=0, double fy=0, int interpolation = INTER_LINEAR);`
   - src – It is the image stored in the first function provided as a parameter in the function main ().
   - dst – It is the resized image.
   - dsize – It is the output image size Size (factor +200, factor + 200)
   - fx –
   - fy –

   fx, fy are the factor changed in the trackbar.

   - Interpolation method:
     - INTER_NEAREST - a nearest-neighbor interpolation
     - INTER_LINEAR - a bilinear interpolation (used by default)
     - INTER_AREA - resampling using pixel area relation. It may be a preferred method for image decimation, as it gives more - free results. But when the image is zoomed, it is similar to the INTER_NEAREST method.
     - INTER_CUBIC - a bicubic interpolation over 4x4 pixel neighborhood
     - INTER_LANCZOS4 - a Lanczos interpolation over 8x8 pixel neighborhood
b. Create a window where the final image will be displayed.
c. Display the final image.

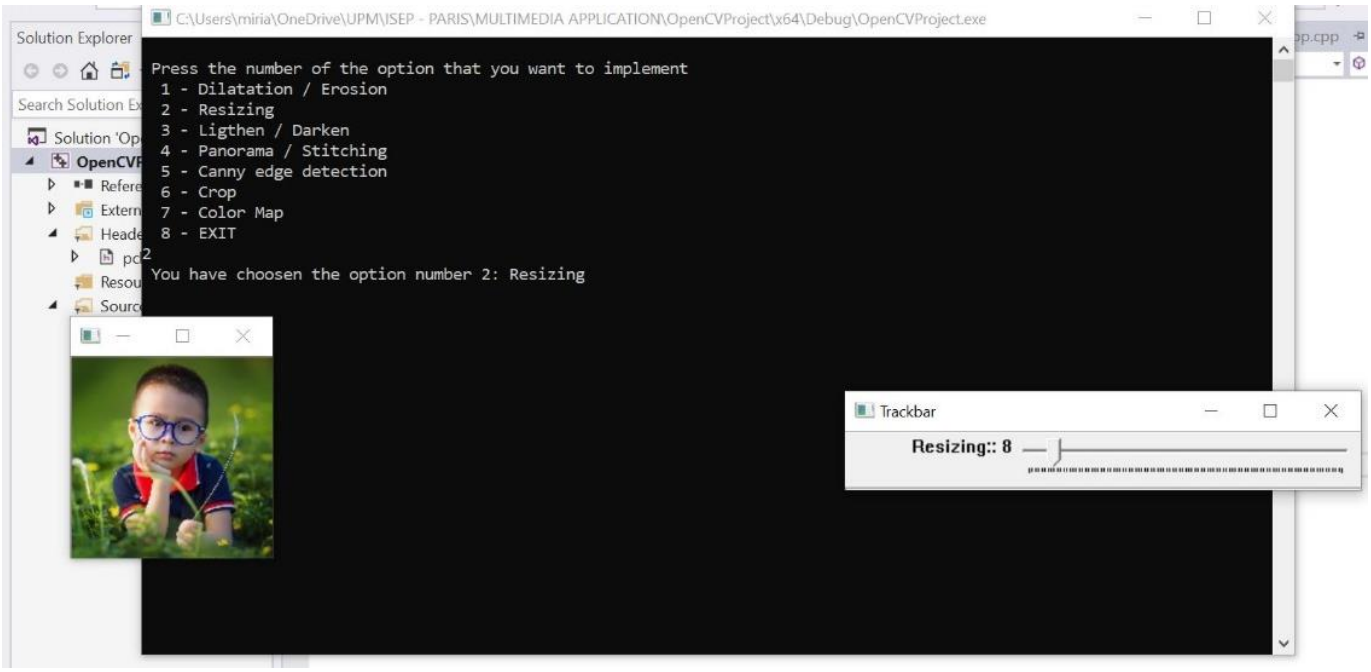


FIgure 3. Resizing a picture

Lighten/Darken

This functionality is built in the same way as the resizing functionality so it has two functions:

1- `void changebright (cv::Mat img);`

a. Store in a local variable the image that is passed as an argument.
b. Create a window where the trackbar will be displayed.
c. Create two trackbars to manage the brightness and the white balance of the lightening/darkening.
   - The brightness has a value between 0 and 300. Between 0 and 99 it will darken the picture and between 101 and 300 it will brighten the picture.
   - The brightness trackbar starts in the value (100) which is for the unmodified picture.
   - The white balance trackbar starts in the value (0) which is for the unmodified picture.
d. Originally Antoine managed to build the functionality without trackbars but did not managed to implement trackbars, Celia corrected it and successfully implemented the trackbars into the functionality.
e. Call to rebright function.

2- `rebright (int, void*);`

a. The double for loops is respectively to pass through each pixel-columns of the picture and then to pass through each pixel of said column.
b. The last for loop is for the picture channels, an option only available if the picture is in color, since we are editing the pixel color value, we need to get the information about the image channel.
c.
```
dstRebright.at<Vec3b>(y, x)[c] = saturate_cast<uchar>((alpha/100)*srcLD.at<Vec3b>(y, x)[c] + beta);
```
This line assigns the modified pixel of the original picture to a pixel (in the same position) for the new image. It uses the alpha (brightness controller) and beta (the white balance controller) to modify the value of the pixel.
The formula itself uses “Vec3b” to access to the color value of the pixel, “saturate_cast<uchar>” is used to convert the value thanks to the calculation which uses the brightness and the white balance. The calculation itself was found on the net.
d. Create a window where the final image will be displayed.
e. Display the final image with the original image.

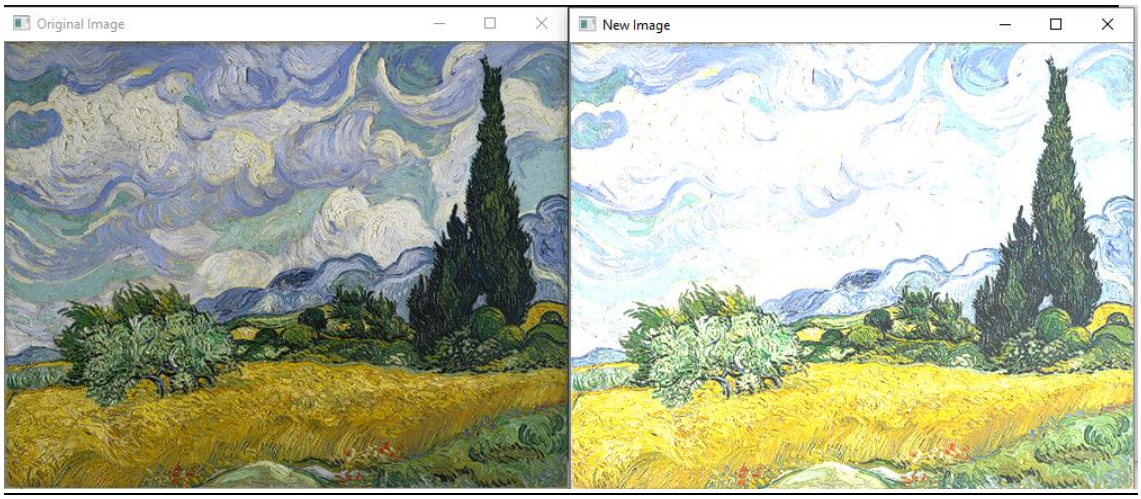


Figure 4. Final result of lightening

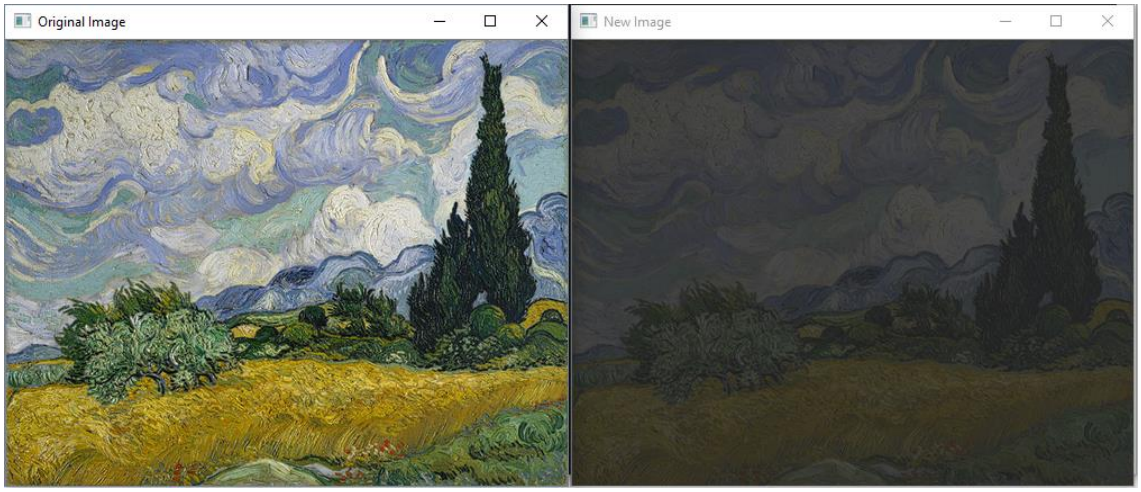


Figure 5. Final result of darkening

Panorama/Stitching

For this functionality we needed to open at least 2 files and possibly work with an entire directory, thus the code organization following that of a main function and 2 support functions.

1- `int stitcher(int argc, char** argv);`
   a. call parseArgs function initializing a vector<cv::Mat> if possible
   b. use opencv stitching algorithm to try and make a panorama out of those files.
   c. Save the resultant panorama.
2- `void getFiles(path p);`
   a. Only called in directory mode
   b. Iterate through the directory and recognize files matching a list of OpenCV supported extentions
   c. Populate the vector of Matrixes depending on what is found.
3- `int parseArgs(int argc, char ** argv);`
   a. verify call syntax (more than 3 args for a directory and more than 4 for file mode)
   b. call getFiles on directory mode.

Figure 6. Original image

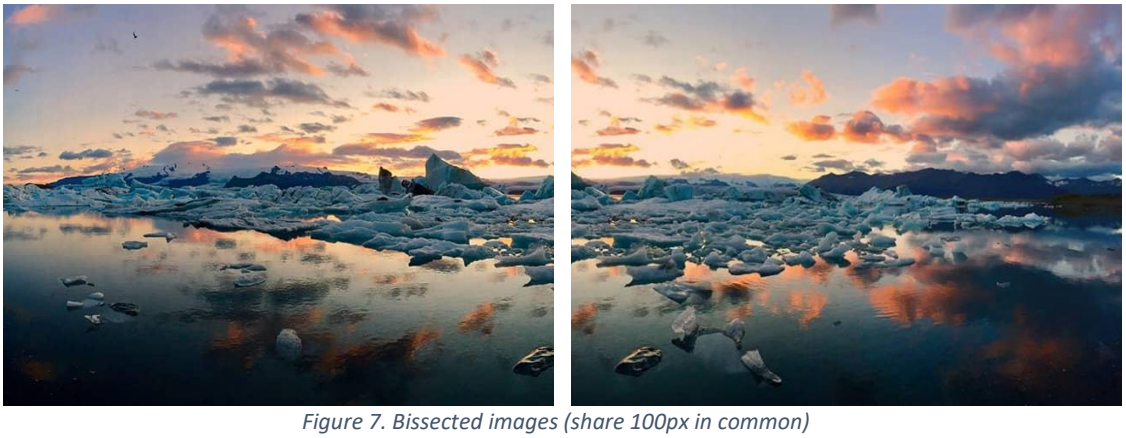


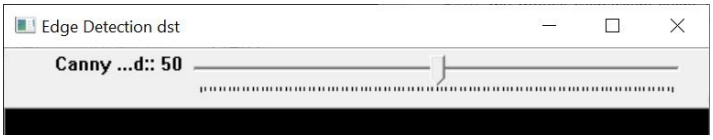
Figure 7. Bissected images (share 100px in common)

Figure 8. Stitched result

Canny Edge detection

For this functionality is been created two functions:

1- `void Canny (cv::Mat img);`
   a. Store in a local variable the image that is passed as an argument by the main function.
   b. Create a matrix destination of the same type and size as src.
   c. Create a window where the trackbar will be displayed. This window will be the same for the final image.
   d. Create a trackbar to manage the factor of canny edge detection.
      - The factor has a value between 0 and 100.
      - The trackbar starts in the mean value (50).



   e. Call to resizing function.

2- `void edgeDetection (int, void*);`
   a. The noise is reduced with a kernel 3x3 with function *blur*.
      - Blurs an image using the normalized box filter,
      - `void blur (InputArray src, OutputArray dst, Size ksize, Point anchor=Point (-1, -1), int borderType=BORDER_DEFAULT);`
        - **src** – input image.
        - **dst** – output image of the same size and type as src.
        - **ksize** – blurring kernel size.
        - **anchor** – anchor point; default value Point (-1, -1) means that the anchor is at the kernel center.
        - **borderType** – border mode used to extrapolate pixels outside of the image.
   b. Call to OpenCV Canny function → `void Canny (InputArray image, OutputArray edges, double threshold1, double threshold2, int apertureSize = 3, bool L2gradient = false);`
      - **image** – input image.
      - **edges** – output edge map; it has the same size and type as image.
      - **threshold1**
      - **threshold2**

        First and second threshold for the hysteresis procedure, they are modified by the trackbar
      - **apertureSize** – aperture size for the Sobel() operator.
      - **L2gradient** – a flag, indicating whether a more accurate L_2 norm $= \sqrt{(dI/dx)^2 + (dI/dy)^2}$ should be used to calculate the image gradient magnitude (L2gradient=true), or whether the default L_1 norm$= |dI/dx| + |dI/dy|$ is enough (L2gradient = false).
   c. Create a window where the final image will be displayed.

d. Display the final image.

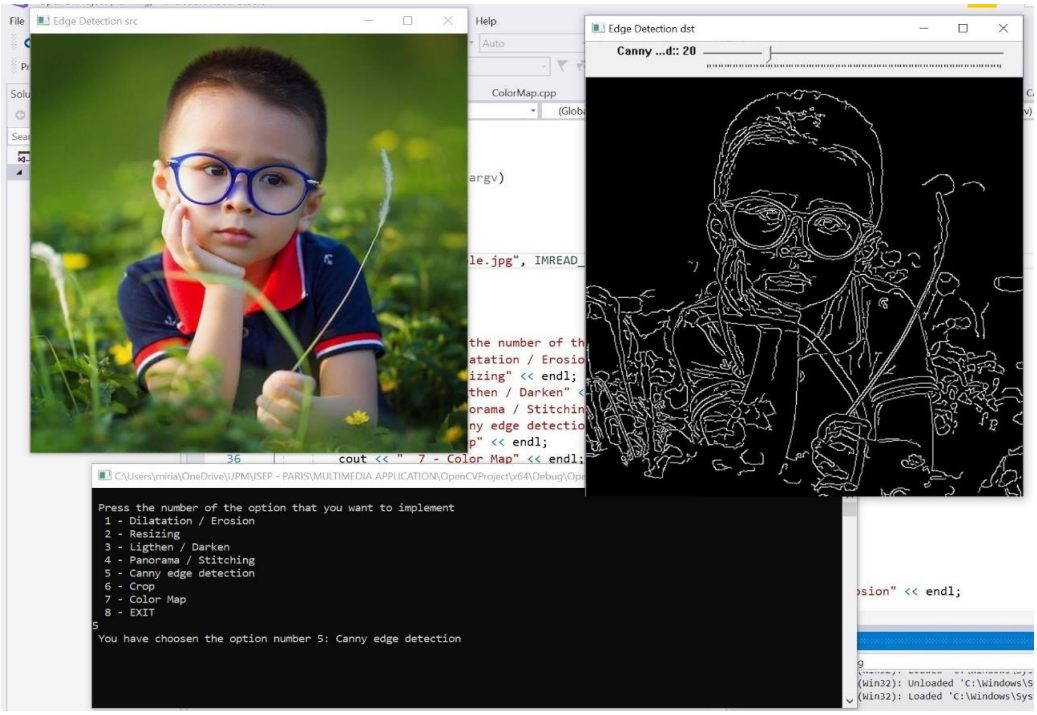


Figure 9. Canny

3. Additional functions

Crop

The idea is cropping an image using the mouse according to the rectangle or square drawn. For this functionality is created two different functions:

1- `void Crop (cv::Mat img);`
   a. Store in a local variable the image that is passed as an argument by the main function.
   b. Create a window in order to display the source image.
   c. When a mouse event occurred is detected with the OpenCV ***setMouseCallback*** and *my_mouse_callback* function is called.
      - void setMouseCallback (const string& **winname**, MouseCallback **onMouse**, void* **userdata**=0)
        - **winname** – Window name where recognize the mouse event.
        - **onMouse** – Mouse callback to the function to process the event.
        - **userdata** – The optional parameter passed to the callback.

2- `void my_mouse_callback (int event, int x, int y, int flags, void* param);`
   a. Process the event detected with the OpenCV ***setMouseCallback*** in a switch case.
   b. In the switch case are processed:

- EVENT_MOUSEMOVE
- EVENT_LBUTTONDOWN
- EVENT_LBUTTONUP

When the left button down is detected, the movement of the mouse is processed and finally when the left button is up is detected the points of the shape drawn are saved as *roi (region of interest)* which is the final image.

c. Create a window where the final image will be displayed.
d. Display the final image.

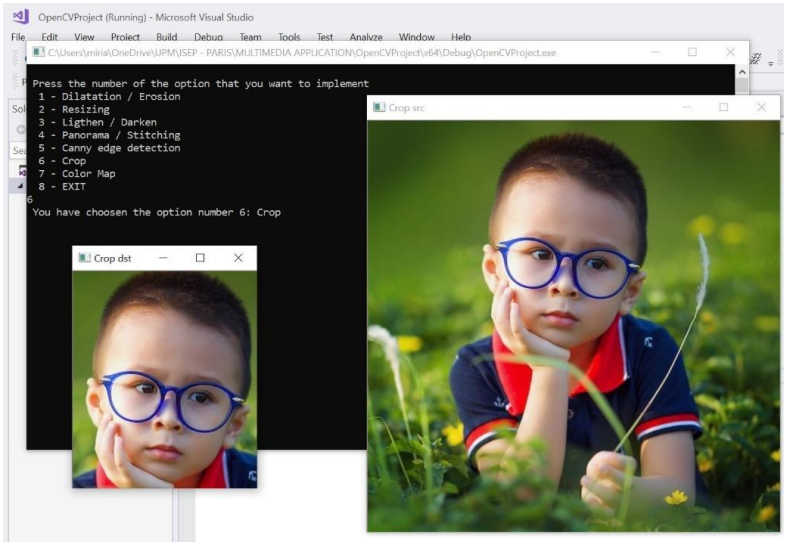


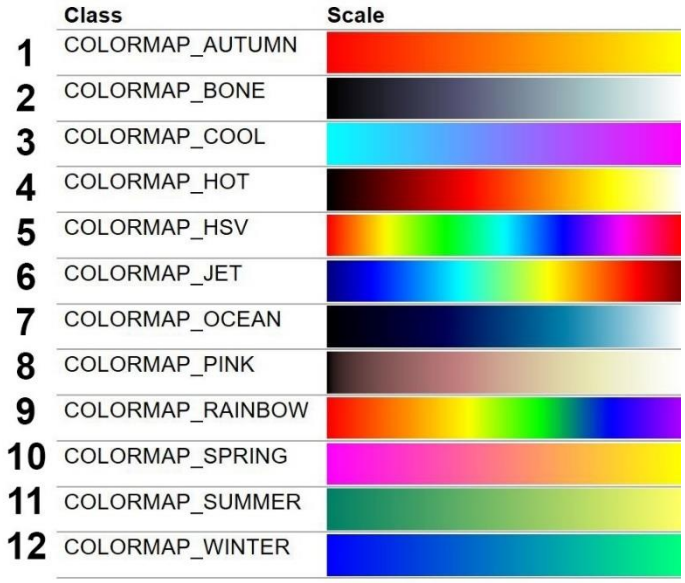
Figure 10. Cropping a picture

ColorMap

The idea is creating some image filters or effects to change the colors like in an editor. For this functionality is created two different functions:

1- `void showInfo ();`
   a. Show the information about the filters with numbers as a main menu.

2- `void ColorMap (cv::Mat img);`
   a. Store in a local variable the image that is passed as an argument by the main function.
   b. Show the menu information calling **`showInfo ()`**.
   c. Ask to the user for the number of the different filters to apply.
   d. If the number zero is chosen, this picture with extended information about the colors will be displayed.



   e. Process the number in a switch case
   f. The color is applied with the OpenCV **applyColorMap** function → `void applyColorMap (InputArray src, OutputArray dst, int colormap);`
      - **src** – input image.
      - **dst** – output image
      - **colormap** – filter applied
   g. Create a window where the final image will be displayed.
   h. Display the final image.

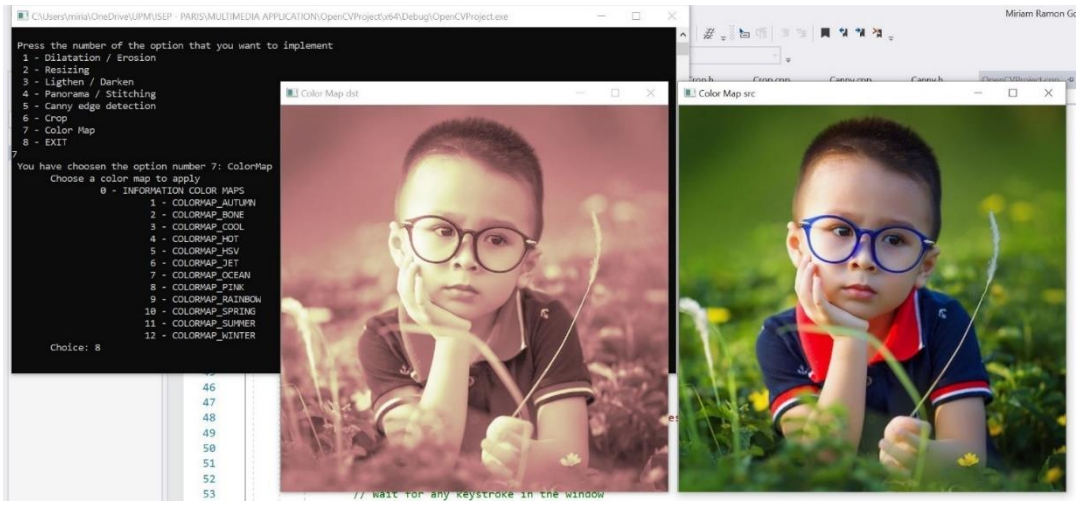


Figure 11. ColorMap with Pink filter.

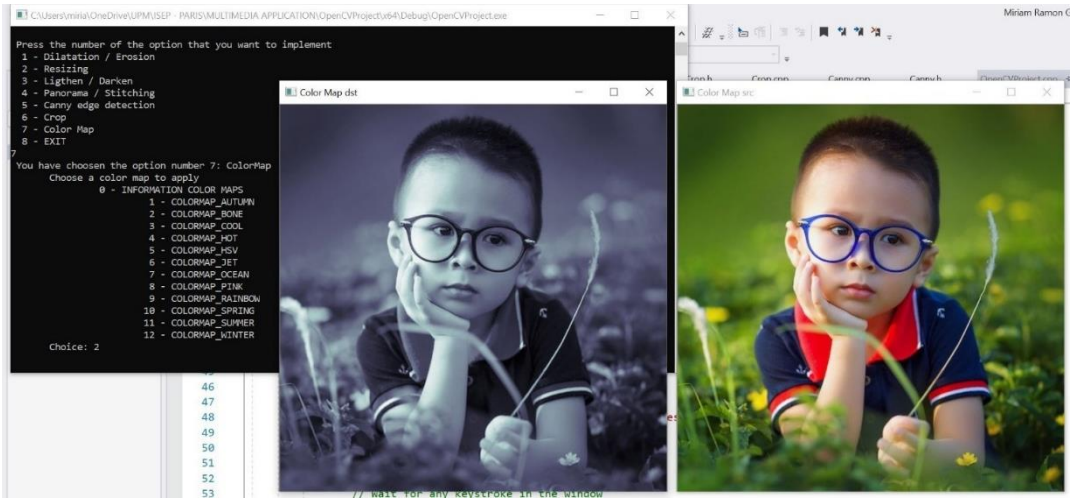


Figure 12. ColorMap with Bone filter.