

Morphological Operations

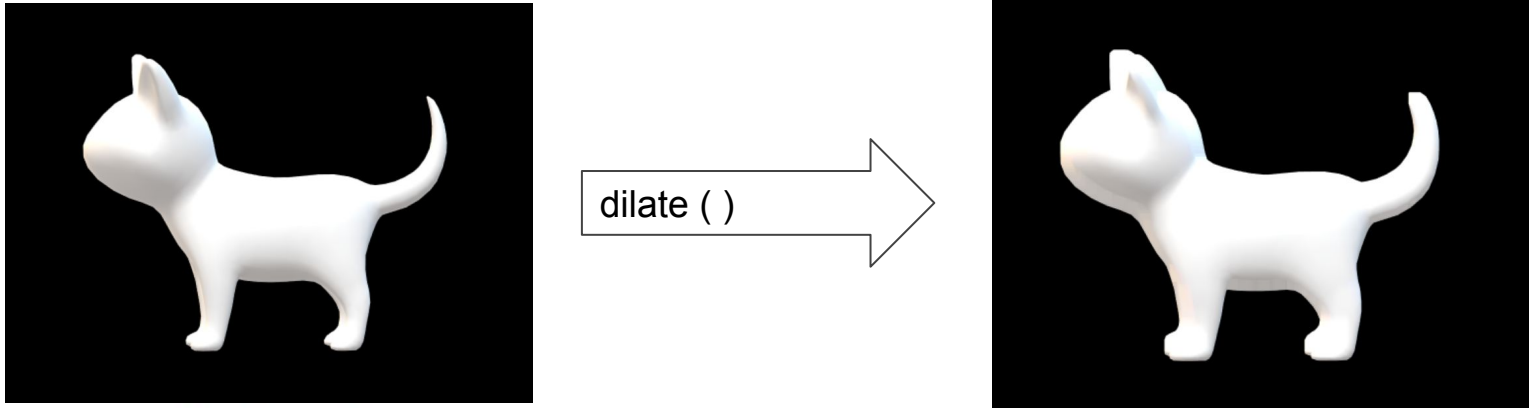
A set of operations that process images based on shapes. Morphological operations apply a structuring element to an input image and generate an output image.

The most basic morphological operations are: Erosion and Dilation. They have a wide array of uses, i.e:

- 1) Removing noise
- 2) Isolation of individual elements and joining disparate elements in an image.
- 3) Finding of intensity bumps or holes in an image

dilate

the function gets a picture and dilate the main regions within the img.



$$: \text{dst}(x, y) = \max_{(x', y') : \text{element}(x', y') \neq 0} \text{src}(x + x', y + y')$$

How does it work?

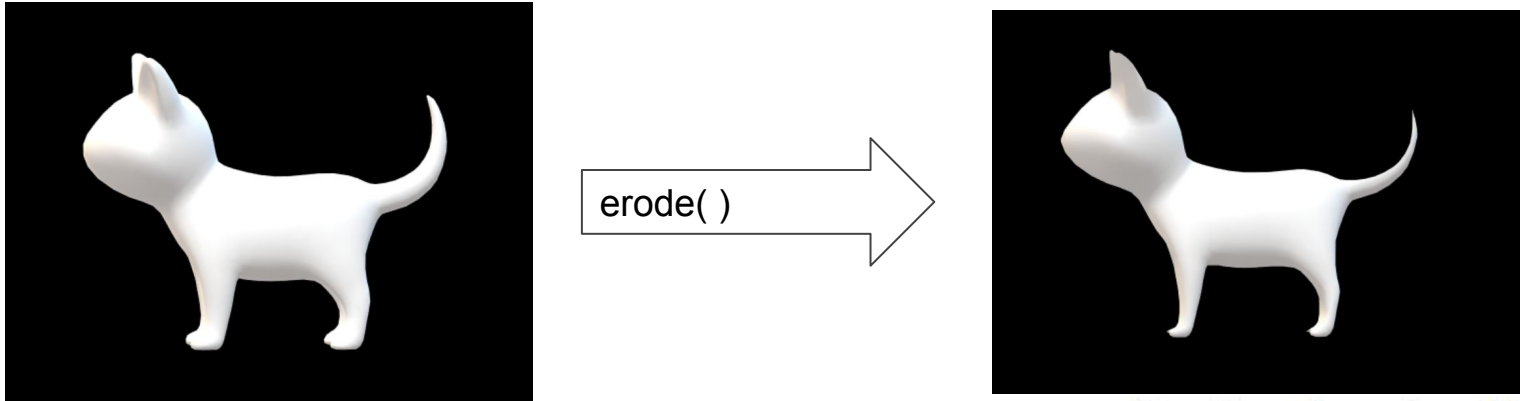
- This operations consists of convolving an image A with some kernel (B), which can have any shape or size, usually a square or circle. The kernel B has a defined *anchor point*, usually being the center of the kernel.
- As the kernel B is scanned over the image, we compute the maximal pixel value overlapped by B
- and replace the image pixel in the anchor point position with that maximal value. As you can deduce, this maximizing operation causes bright regions within an image to "grow" (therefore the name *dilation*).



```
void Dilation(int, void*)  
{  
    int dilation_size = 3;  
    Mat element = getStructuringElement(MORPH_RECT,  
        Size(2 * dilation_size + 1, 2 * dilation_size + 1),  
        Point(dilation_size, dilation_size));  
  
    dilate(src, dilation_dst, element);  
    imshow("Dilation Demo", dilation_dst);  
}
```

erode

the function gets a picture and erodes the main regions within the img.



$$\text{dst}(x, y) = \min_{(x', y') : \text{element}(x', y') \neq 0} \text{src}(x + x', y + y')$$

How does it work?

- This operation is the sister of dilation. It computes a local minimum over the area of given kernel.
- As the kernel B is scanned over the image, we compute the minimal pixel value overlapped by B and replace the image pixel under the anchor point with that minimal value.



```
void Erosion(int, void*)  
{  
    int erosion_size = 3;  
    Mat element = getStructuringElement(MORPH_RECT,  
        Size(2 * erosion_size + 1, 2 * erosion_size + 1),  
        Point(erosion_size, erosion_size));  
  
    erode(src, erosion_dst, element);  
    imshow("Erosion Demo", erosion_dst);  
}
```

morphologyEx()

the function `morphologyEx()` : `img_src`, `img_dst` and enum of the morph type: open or close.

`MORPH_OPEN` is the obtained by the erosion of an image followed by a dilation. it cleans noises out the item.

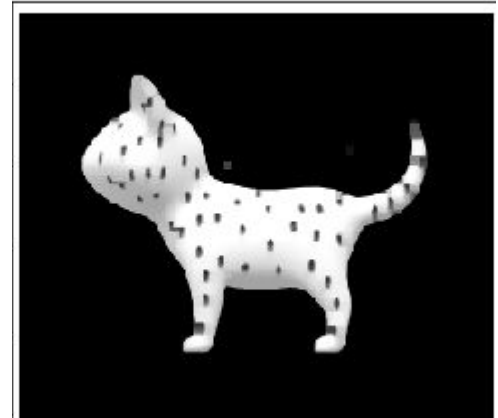
`MORPH_CLOSE` is the obtained by the dilation of an image followed by an erosion. it cleans noises within the item.



MORPH_OPEN



MORPH_OPEN ()



`dst = open(src, element) = dilate(erode(src, element))`

```
void Open(int, void*)  
{  
    int open_size = 3;  
    Mat element = getStructuringElement(MORPH_RECT,  
        Size(2 * open_size + 1, 2 * open_size + 1),  
        Point(open_size, open_size));  
  
    morphologyEx(src, open_dst, MORPH_OPEN, element);  
    imshow("Open Demo", open_dst);  
}
```

MORPH_CLOSE



MORPH_CLOSE ()



```
dst = close(src, element) = erode(dilate(src, element))
```

```
void Close(int, void*)  
{  
    int close_size = 3;  
    Mat element = getStructuringElement(MORPH_RECT,  
        Size(2 * close_size + 1, 2 * close_size + 1),  
        Point(close_size, close_size));  
  
    morphologyEx(src, close_dst, MORPH_CLOSE, element);  
    imshow("Close Demo", close_dst);  
}
```

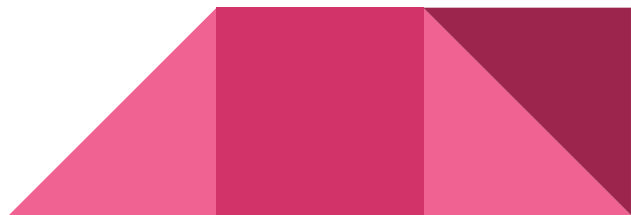
Top Hat

$$\text{Tophat} = \text{src image} - \text{open}$$

The pixels that would be removed by "*opening*"



```
void Morphology_Tophat()  
{  
    Mat element = getStructuringElement(0, Size(5,5));  
    morphologyEx(src, dst, MORPH_TOPHAT, element);  
    imshow(window_name, dst);  
}
```



Top Hat



MORPH_TOPHAT



Src image



—

open image



=

tophat



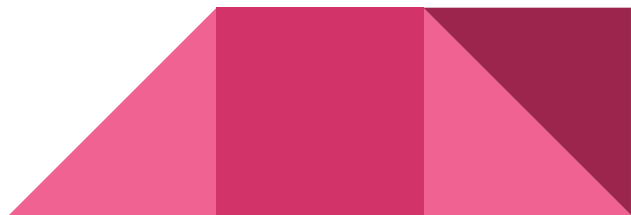
BlackHat

BlackHat = closing – src image

The pixels that would be added by "*closing*"



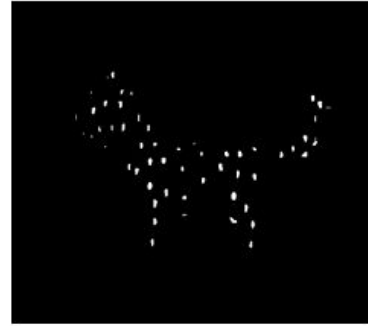

```
void Morphology_Blackhat()  
{  
    Mat element = getStructuringElement(0, Size(5,5));  
    morphologyEx(src, dst, MORPH_BLACKHAT, element);  
    imshow(window_name, dst);  
}
```



Black Hat



MORPH_BLACKHAT



close image



—

src image



=

blackhat



Making Your Own Kernel

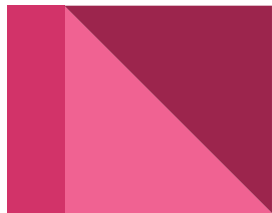
1. Using the function `getStructuringElement()`, pre-defined shapes.
2. Construct an arbitrary binary mask yourself and use it as the structuring element.

How does it work?

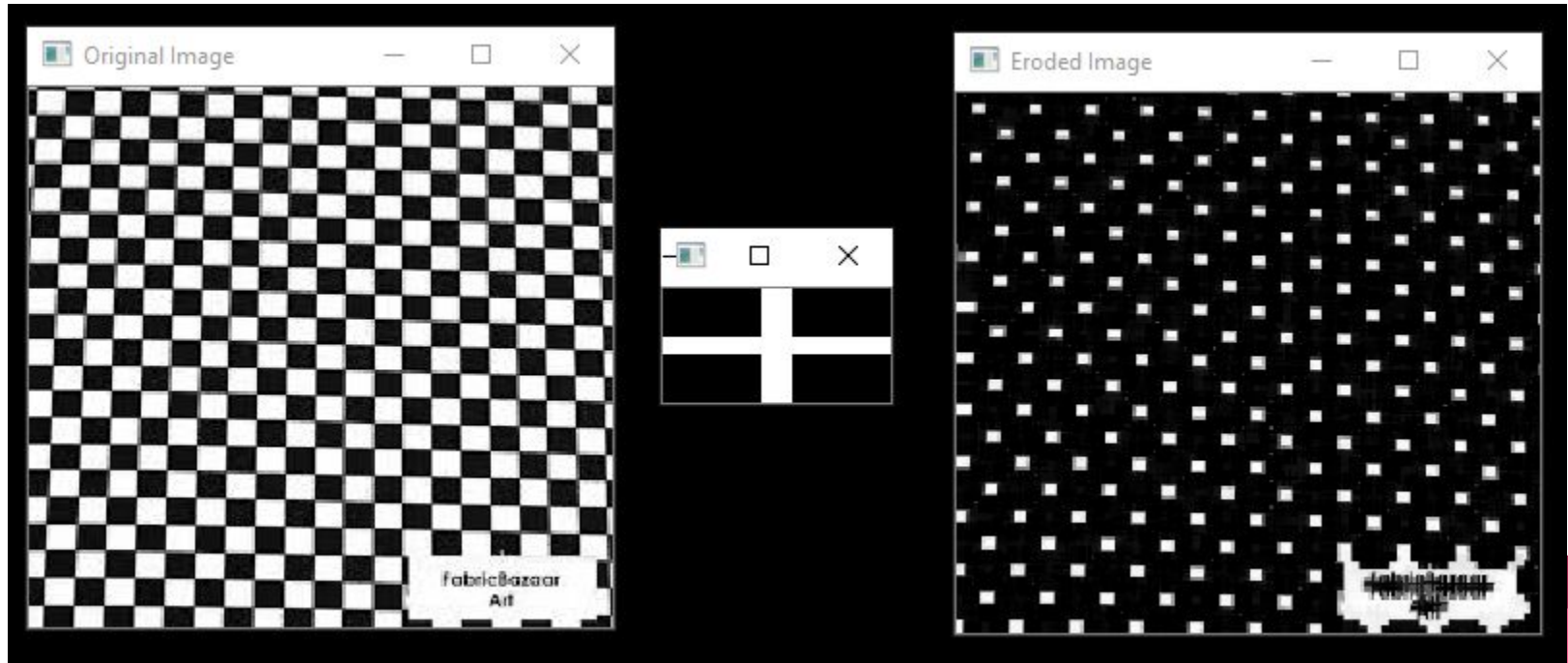
1. We send the function a shape, size and a anchor. It returns an array of that size with ones defining the shape. The anchor defines the center pixel of the structuring element, the pixel being processed.
2. We manually create a structuring elements by creating and manipulating Mat's.

Use case

Create kernels of common shapes, such as lines, diamonds, disks, periodic lines... for any filter.

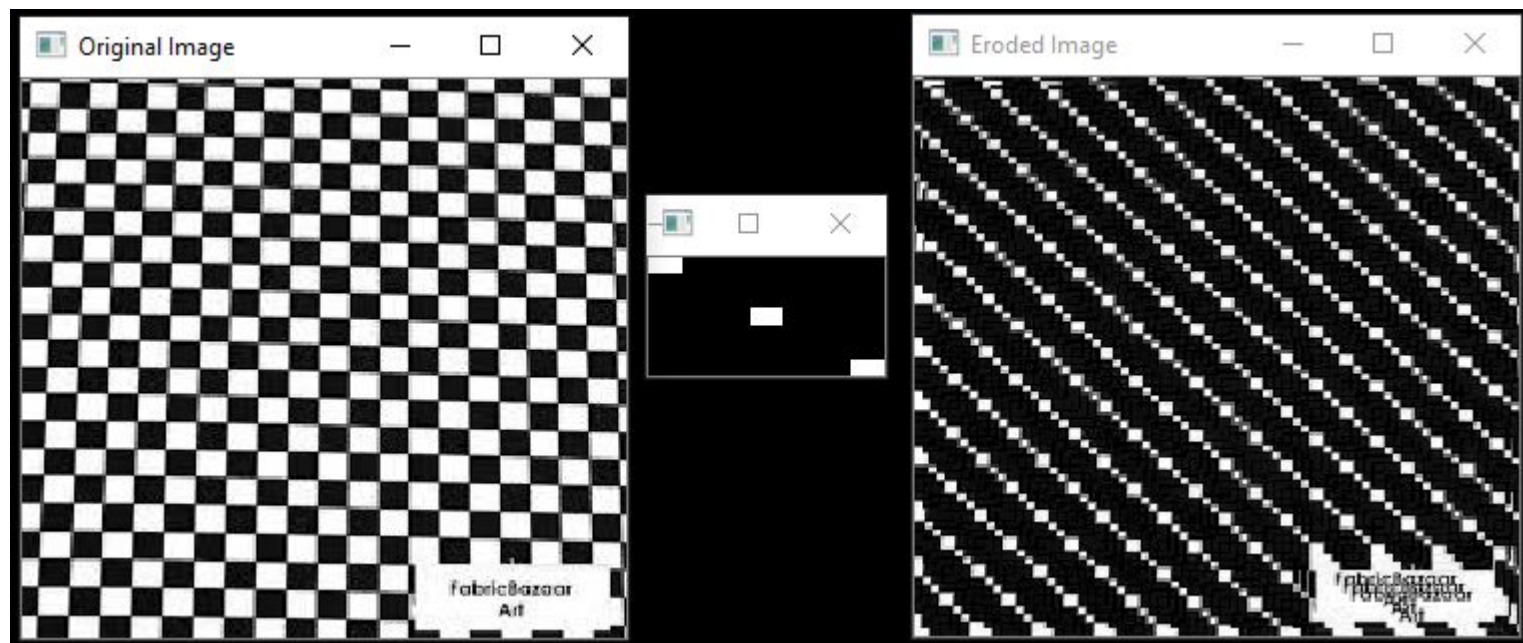


getStructuringElement()



```
void exampleGetStructuringElement()  
{  
    Mat image = imread("checkers_before_filter.jpg", IMREAD_GRAYSCALE);  
    Mat kernel = getStructuringElement(MORPH_CROSS, Size(7, 7)) * 255;  
    Mat eroded;  
  
    erode(image, eroded, kernel);  
  
    namedWindow("Original Image", WINDOW_FREERATIO);  
    namedWindow("Eroded Image", WINDOW_FREERATIO);  
    namedWindow("Kernel", WINDOW_FREERATIO);  
    imshow("Original Image", image);  
    imshow("Eroded Image", eroded);  
    imshow("Kernel", kernel);  
  
    waitKey(0); //waits until something is pressed  
}
```

Construct an arbitrary binary mask yourself




```
void exampleOwnKernel()
{
    Mat image = imread("checkers_before_filter.jpg", IMREAD_GRAYSCALE);

    Mat kernel = Mat::zeros(Size(7, 7), CV_8U);
    kernel.at<uint8_t>(0, 0) = 255;
    kernel.at<uint8_t>(3, 3) = 255;
    kernel.at<uint8_t>(6, 6) = 255;

    Mat eroded;
    erode(image, eroded, kernel);

    namedWindow("Original Image", WINDOW_FREERATIO);
    namedWindow("Eroded Image", WINDOW_FREERATIO);
    namedWindow("Kernel", WINDOW_FREERATIO);
    imshow("Original Image", image);
    imshow("Eroded Image", eroded);
    imshow("Kernel", kernel);


    waitKey(0);
}
```


Extract horizontal and vertical lines by using morphological operations

How does it work?

We create kernels that will filter out only the horizontal and vertical lines and subtract those results from the image.

Use case

- Separate music notes from a music sheet.
 - Separate numbers from grid.
 - Etc.
- 

 image — □ ×

9	1	3	8	2	6	6	5	9	8	0	7	7	2	9
0	0	4	1	1	7	4	4	9	0	6	4	4	2	6
9	0	9	7	0	9	2	1	0	2	4	6	0	2	2
6	2	8	0	8	9	3	5	8	0	5	5	8	7	3
6	3	9	6	9	5	3	4	7	1	9	0	2	9	5
3	3	0	0	9	9	6	8	4	1	5	6	0	7	7
0	2	9	6	3	9	2	1	3	1	3	4	0	1	2
3	0	1	2	1	3	8	9	7	0	9	9	6	7	3
8	8	2	5	0	7	2	0	0	5	6	7	3	1	4
6	1	3	3	0	2	8	3	0	1	2	3	0	7	7
2	7	5	8	1	0	0	3	0	4	4	4	5	1	6

A screenshot of a terminal window. The title bar at the top shows a green icon, the letter 'h', and standard window controls (minimize, maximize, close). The terminal area is black with a white prompt character at the top left, ready for input.

Numbers															—	□	×
9	1	3	8	2	6	6	5	9	8	0	7	7	2	9			
0	0	4	1	1	7	4	4	9	0	6	4	4	2	6			
9	0	9	7	0	9	2	1	0	2	4	6	0	2	2			
6	2	8	0	8	9	3	5	8	0	5	5	8	7	3			
6	3	9	6	9	5	3	4	7	1	9	0	2	9	5			
3	3	0	0	9	9	6	8	4	1	5	6	0	7	7			
0	2	9	6	3	9	2	1	3	1	3	4	0	1	2			
3	0	1	2	1	3	8	9	7	0	9	9	6	7	3			
8	8	2	5	0	7	2	0	0	5	6	7	3	1	4			
6	1	3	3	0	2	8	3	0	1	2	3	0	7	7			
2	7	5	8	1	0	0	3	0	4	4	4	5	1	6			

```
void exampleVerticalAndHorizontalLines()
{
    Mat image = imread("grid_before_filter.png", IMREAD_GRAYSCALE);
    image = ~image; //switches between black and white

    Mat horizontalSE = getStructuringElement(MORPH_RECT, Size(image.cols / 15, 1));
    Mat verticalSE = getStructuringElement(MORPH_RECT, Size(1, image.rows / 11));

    Mat hErode, vErode;

    erode(image, hErode, horizontalSE);
    erode(image, vErode, verticalSE);

    imshow("image", image);

    image = image - hErode - vErode;

    imshow("h", hErode);
    imshow("v", vErode);
    imshow("Numbers", image);

    waitKey(0);
}
```

Color Detection

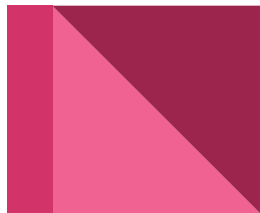
Detect an object from an image with color based methods.

How does it work?

We perform thresholding on each frame from the video using the `inRange()` function which returns a black and white image based on the range of pixel values in the HSV colorspace.

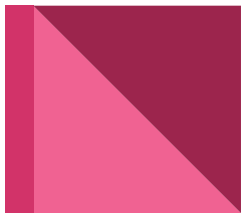
Use case

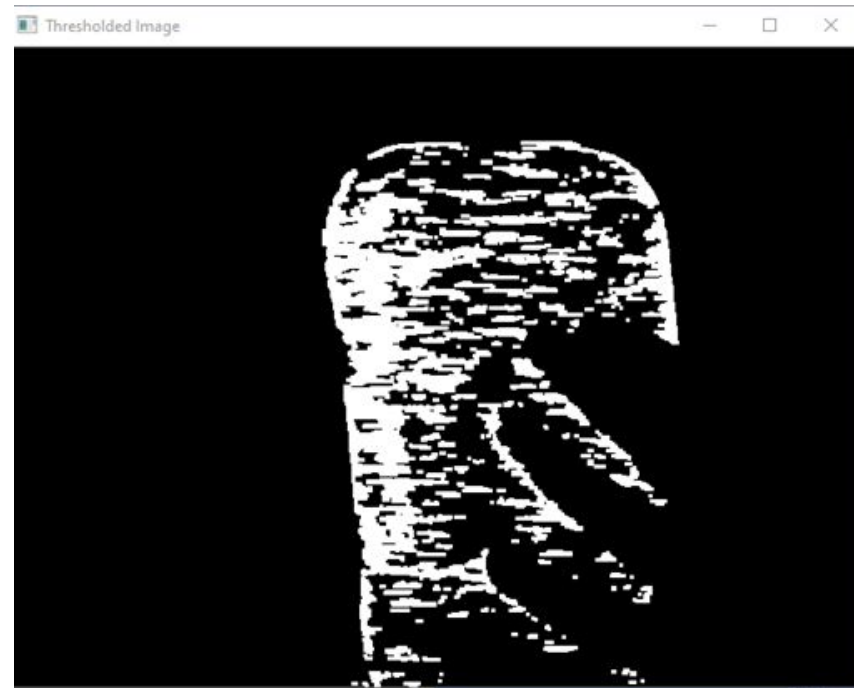
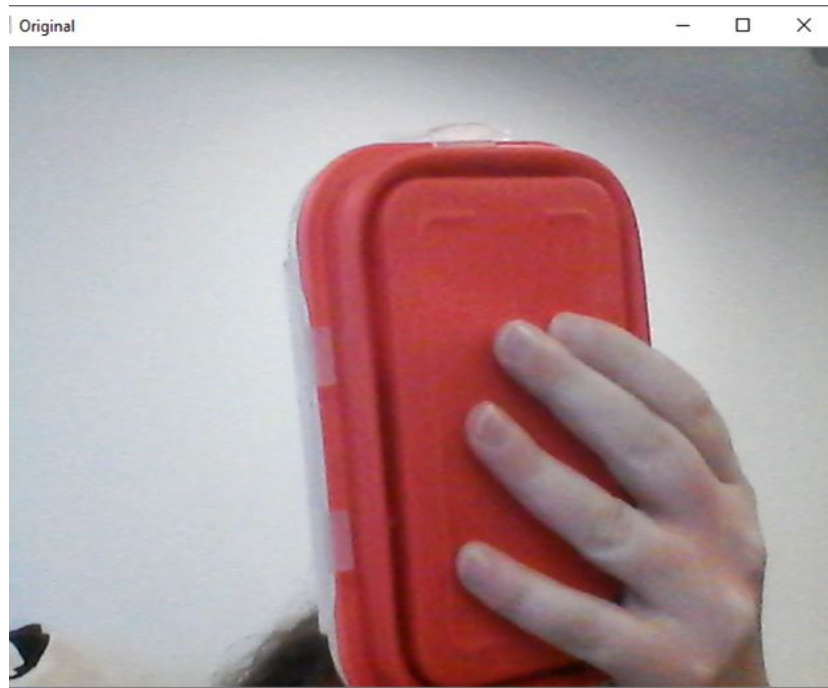
- Identify red traffic lights.
- Identify a colored object..



HSV

- More closely aligned with the way human vision perceives color-making attributes.
- Color space is also consists of 3 matrices, HUE, SATURATION and VALUE.
- HUE represents the color: 0 – 179.
- SATURATION represents the amount to which that respective color is mixed with white: 0 – 255.
- VALUE represents the amount to which that respective color is mixed with black: 0 – 255.





```
while (true)
{
    Mat imgOriginal;

    bool bSuccess = cap.read(imgOriginal); // read a new frame from video

    if (!bSuccess) //if not success, break loop
    {
        cout << "Cannot read a frame from video stream" << endl;
        break;
    }

    Mat imgHSV;

    cvtColor(imgOriginal, imgHSV, COLOR_BGR2HSV); //Convert the captured frame from BGR to HSV

    Mat imgThresholded;

    inRange(imgHSV, Scalar(iLowH, iLowS, iLowV), Scalar(iHighH, iHighS, iHighV), imgThresholded); //T

    dilate(imgThresholded, imgThresholded, getStructuringElement(MORPH_RECT, Size(3, 3)));

    imshow("Thresholded Image", imgThresholded); //show the thresholded image
    imshow("Original", imgOriginal); //show the original image

    if (waitKey(30) == 27) //wait for 'esc' key press for 30ms. If 'esc' key is pressed, break loop
    {
        cout << "esc key is pressed by user" << endl;
        break;
    }
}
```