Introduction to Visual Computing

Assignment# 8

Basics of Image Processing

April 6, 2020

Description

The next step of your project is to control the board of the game with a tangible board. To this end, we need to detect a physical object (a Lego green board) from a webcam image stream, localize the four corners, use them to compute the 3D orientation of the physical board and finally, apply this orientation to the virtual board of your game.

This work is spread over three weeks. This week focuses on the basics of image processing: how to preprocess a still image so that the edges of the board stand out.

Objectives

Implement a first set of image processing filters: Gaussian blur, edge detection, thresholding.

Specific Challenges

Manipulating pixel arrays, understanding colour representation, understanding the idea of kernels.

Preliminary steps

- From Moodle, fetch all the pictures of the Lego board that we will use today.

Part I

Colour segmentation

Step 1 – Display an image

Processing has direct support for images. Create a new Processing Sketch and use the following code to display the image `board1.jpg`. (to add the image to your project go to Sketch->Add File)

```
PImage img;

void settings() {
    size(1600, 600);

}
void setup() {
    img = loadImage("board1.jpg");
    noLoop(); // no interactive behaviour: draw() will be called only once.
}

void draw() {
    image(img, 0, 0);
}
```

Step 2 – Accessing pixel values

By default Processing uses Red, Green, Blue (RGB) colormode, each channel having a range of integer values between 0 and 255. An image stores the color value of the pixels in the `pixels[]` array. In order to read the pixel values, you have to load them into the `pixels[]` array by calling the function

`img.loadPixels()`. After you can get the value of a pixel directly accessing the array `img.pixels[y*width+x]`. The returned value is of `color` datatype, so to access the value of the single Red, Green, Blue channels you have to use the function `red()`, `green()`,`blue()`. To set the value of a pixel to a specific color, write `img.pixels[y*width+x]=color(r,g,b)`. Finally, to apply the changes call the function `img.updatePixels()`. If you want to modify an image while keeping a copy of the original, use the function `img.copy()`. This will perform a **deep copy** of the image.

Note

Remember that if you just write `PImage img2=img`, the two variables `img` and `img2` will point at the same image, so changes applied on one variable will affect the other one as well.

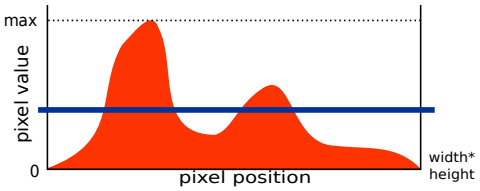
In the following example we will make the even horizontal lines of pixels green.

```
void draw() {
  image(img, 0, 0);//show image
  PImage img2 = img.copy();//make a deep copy
  img2.loadPixels();// load pixels
  for (int x = 0; x < img2.width; x++)
    for (int y = 0; y < img2.height; y++)
      if (y%2==0)
        img2.pixels[y*img2.width+x] = color(0, 255, 0);
  img2.updatePixels();//update pixels
  image(img2, img.width, 0);
}
```

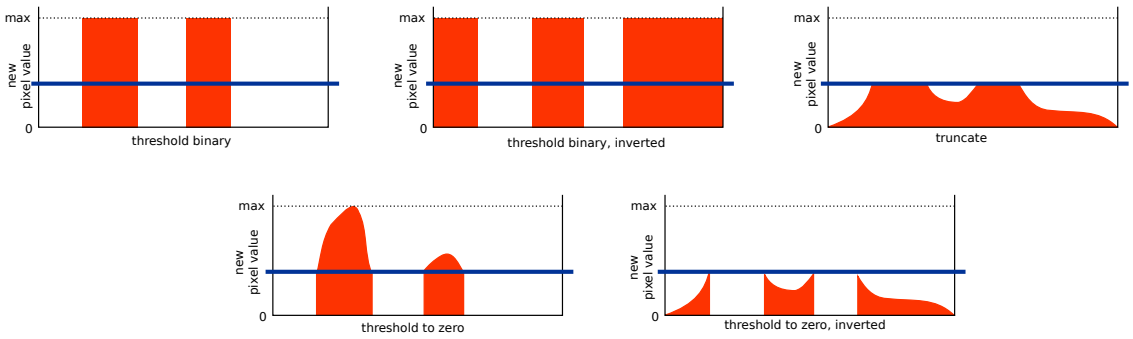
Step 3 – Basic manipulation of the image: thresholding

Thresholding an image is the most basic image processing operation: it consists of rejecting certain pixels whose value does not match a specific criterion (like a minimum level of brightness).

The following figure represents, for instance, the intensity profile of a gray-scale image[1], with pixel values ranging from 0 to a maximum value; the blue line represents a fixed threshold:



Five standard thresholding operations can be performed, resulting, in the case above, in the following profiles:



Implement and apply the first two operations on your image, filtering pixels by their intensity with a fixed threshold of 128. Do so by iterating over all pixels:

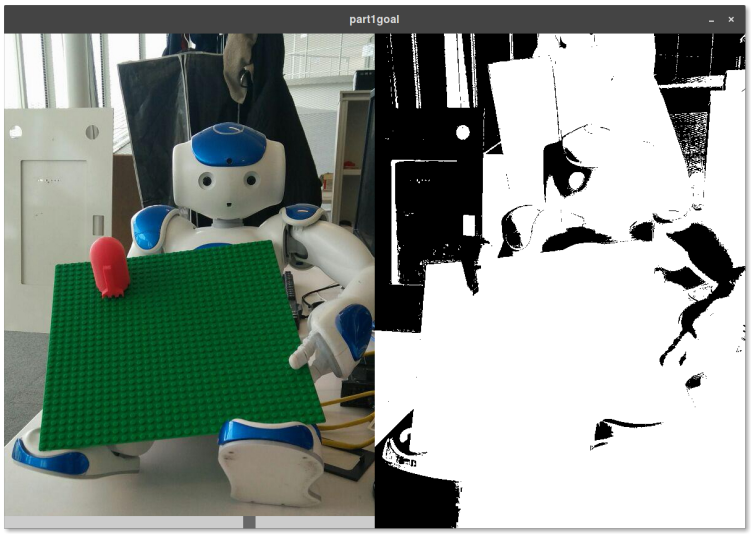
```
PImage threshold(PImage img, int threshold){
        // create a new, initially transparent, 'result' image
        PImage result = createImage(img.width, img.height, RGB);
        for(int i = 0; i < img.width * img.height; i++) {
            // do something with the pixel img.pixels[i]
        }
    return result;
}
```

Use the `brightness()` function to obtain a pixel's intensity, and check the documentation of the `color()` function to see how to set pixels' values. This function will be useful for the rest of the project!

[1] Gray-scale image in RGB colormode has the same value in all the three channels.

Step 4 – A bit of user interface

Use the `HScrollbar` class that you saw during week 7 to add a scrollbar to the application that sets the threshold level:



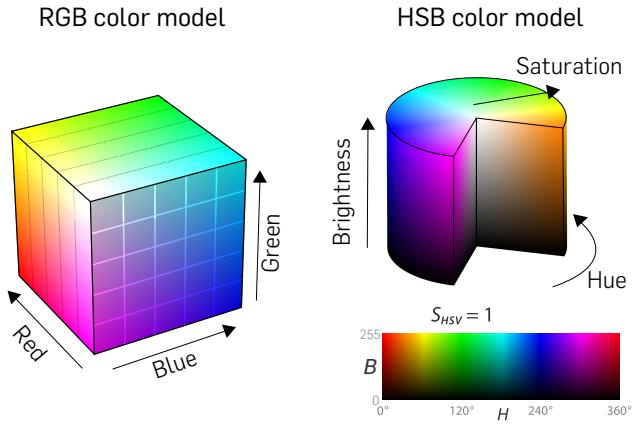
The following code sample shows how to use the scrollbar:

```
HScrollbar thresholdBar;

void settings() {
  size(800, 600);
}

void setup() {
  thresholdBar = new HScrollbar(0, 580, 800, 20);

  //...
  //noLoop(); you must comment out noLoop()!
}

void draw() {

  //...

  thresholdBar.display();
  thresholdBar.update();

  println(thresholdBar.getPos()); // getPos() returns a value between 0 and 1

}
```

Step 5 – Colour thresholding

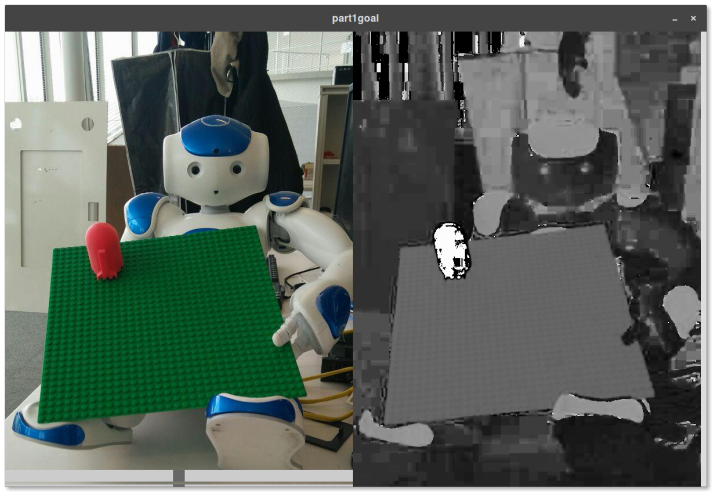
Many image processing algorithms need to select a specific colour in the image. Doing so in the Red-Green-Blue colour space is difficult because it does not effectively separate the **hue** of a colour from its **brightness** and **saturation** (how dull or vivid a colour looks).

On the other hand, the **HSB** colour space (or similar spaces like HSV) better represents the *semantics* of a colour: its hue.

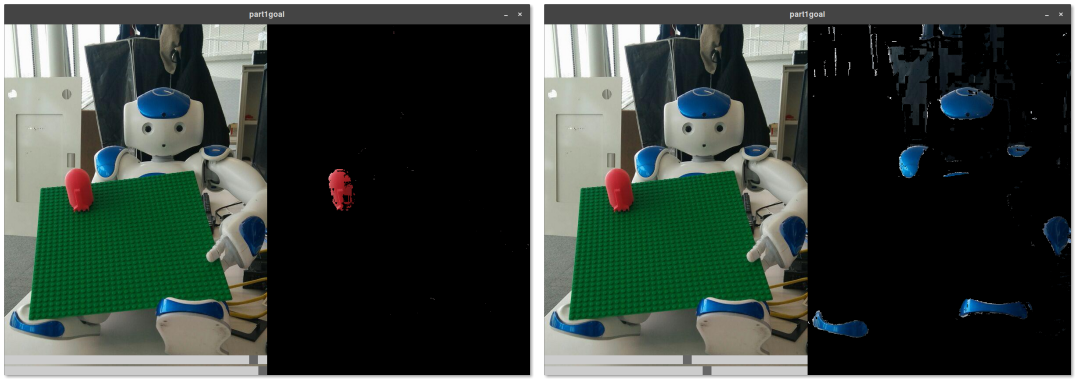


Hue is often expressed as an angle in degree. Processing however represents it as a value between 0 and 255 (likewise for saturation and brightness). In this step, you will have to:

- Using the function `hue()`, transform the image to display its hue map;

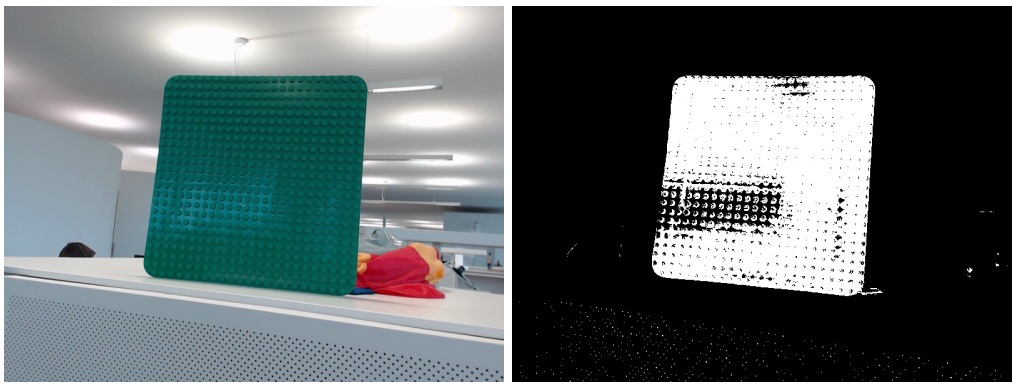


- Then, modify your application to select only pixels within a specific range of hue with two scrollbars. For instance, in the example below, only the red zones of the image have been selected on the left, only the blue on the right:



- Finally, using the `saturation()` and `brightness()` methods, implement a thresholding function on the three channels HSB, that takes min and max input parameters for each channels, such as: `thresholdHSB(PImage img, int minH, int maxH, int minS, int maxS, int minB, int maxB)`). This function will be used for the rest of the project!
- Test your function on the image `board1.jpg` using these thresholding values: minH=100, maxH=200, minS=100 , maxS =255, minB=45, maxB=100 (Pixels are set to white even if their values are equal to these extremes). Is it equal to the image *board1Thresholded.bmp*? Compare the two images with this function:

```
boolean imagesEqual(PImage img1, PImage img2){
    if(img1.width != img2.width || img1.height != img2.height)
            return false;
    for(int i = 0; i < img1.width*img1.height ; i++)
            //assuming that all the three channels have the same value
        if(red(img1.pixels[i]) != red(img2.pixels[i]))
                return false;
    return true;
}
```



Part II

Convolution of kernels

The convolution of kernels on images is the fundamental operation performed in image processing. A **kernel** (also called **convolution matrix** or **mask**) is a small matrix. By applying it to the pixel array of the image through a **convolution**, many effects can be achieved, including **blurring** and **edge detection**. By combining several convolutions, complex operations like **morphological transformations** (erosion, dilation or skeletonization of the image, etc.) can be achieved.

In this second part, you will implement a kernel convolution first to blur the image, then to detect edges using the Scharr operator. The implementations assume that the input image is in gray-scale, hence get the value of each pixel using the function `brightness()`.

Step 1 – Convolution of a kernel

As said, a kernel is a small matrix. The *convolution* operation can be understood as: for each pixel of the image, we compute its updated value as the sum of surrounding pixels weighted by the kernel values.

Consider the 3×3 kernels $kernel1$ and $kernel2$ below, and try to guess the effect they achieve:

$$kernel1 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$kernel2 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

To verify your intuition, implement the convolution by writing a function `convolute(img)` that returns a new image with a convolution applied (assume that the kernel size is 3×3):

Note

Pay attention to the borders! To apply the kernel, you need to keep a one-pixel wide border around the image.

```
PImage convolute(PImage img) {

    float[][] kernel = { { 0, 0, 0 },
                         { 0, 2, 0 },
                         { 0, 0, 0 }};
```

```
  float normFactor = 1.f;

  // create a greyscale image (type: ALPHA) for output
  PImage result = createImage(img.width, img.height, ALPHA);

  // kernel size N = 3
  //
  // for each (x,y) pixel in the image:
  //     - multiply intensities for pixels in the range
  //       (x - N/2, y - N/2) to (x + N/2, y + N/2) by the
  //       corresponding weights in the kernel matrix
  //     - sum all these intensities and divide it by normFactor
  //     - set result.pixels[y * img.width + x] to this value

  return result;
}
```

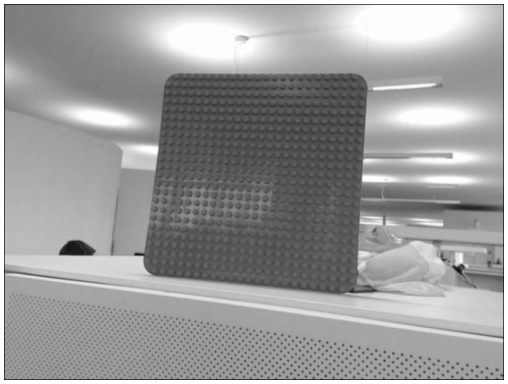
Test your code with the kernels `kernel1` and `kernel2`. Some convolution operations require a normalization factor (`normFactor`) different from 1, for example to make sure that output values remain in a given range.

Step 2 – Gaussian blur

The following kernel is a 3×3 Gaussian blur (*Gaussian* because the weights follow a Gaussian distribution).

$$gaussianKernel = \begin{bmatrix} 9 & 12 & 9 \\ 12 & 15 & 12 \\ 9 & 12 & 9 \end{bmatrix}$$

Since this filter performs a weighted average, the normalization factor is equal to the sum of the kernel values (99). Use this kernel to perform a Gaussian blur on the test image *board1.jpg* and compare the result with *board1Blurred.bmp*.

Step 3 – Edge detection: the Scharr operator

Edge detection can be performed by first computing at each pixel the approximated gradient magnitude as a measure of edge strength and then selecting the "strong" edges. A first approximation of the x-derivative and y-derivative can be obtained by applying the following kernels

$$hKernel = \begin{bmatrix} 0 & +1 & 0 \\ 0 & 0 & 0 \\ 0 & -1 & 0 \end{bmatrix}$$

$$vKernel = \begin{bmatrix} 0 & 0 & 0 \\ +1 & 0 & -1 \\ 0 & 0 & 0 \end{bmatrix}$$

A better approximation can be achieved using the **Sobel operator** or, in case of 3x3 kernel, the **Scharr operator**. The **Scharr operator** is one of the most basic kernels used in edge detection algorithms. It relies on the combination of these two convolution kernels (normFactor=1.0):

$$hKernel = \begin{bmatrix} +3 & +10 & +3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{bmatrix}$$

$$vKernel = \begin{bmatrix} +3 & 0 & -3 \\ +10 & 0 & -10 \\ +3 & 0 & -3 \end{bmatrix}$$

The algorithm works as follow.

For each (x, y) pair:

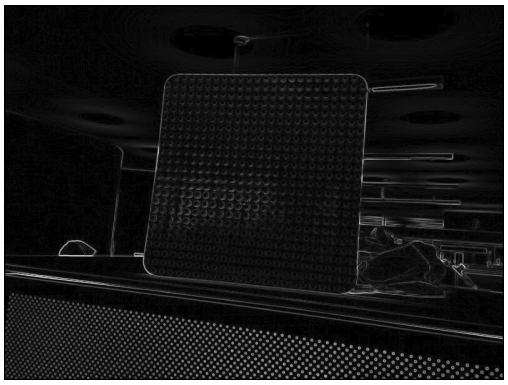
- apply the vertical and horizontal kernels as you did for the Gaussian blur above, and store the sum of intensities into two variables `sum_h` and `sum_v`.
- compute the compound sum as an Euclidean distance `sum=sqrt(pow(sum_h, 2) + pow(sum_v, 2))`. This is the gradient magnitude approximation.
- store this sum into a buffer (defined for instance as `float[] buffer = new float[img.width * img.height]`).
- store as well the maximum value found.

Then, iterate over the buffer, and store in the result image the gradient magnitude, rescaling the values from 0 to `max` to the color range 0-255.

```
PImage scharr(PImage img) {

    float[][] vKernel = {
                {  3, 0, -3  },
                { 10, 0, -10 },
                {  3, 0, -3  } };
```

```
  float[][] hKernel = {
              {  3, 10, 3 },
              {  0, 0, 0 },
              { -3, -10, -3 } };


  PImage result = createImage(img.width, img.height, ALPHA);

  // clear the image
  for (int i = 0; i < img.width * img.height; i++) {
    result.pixels[i] = color(0);
  }

  float max=0;
  float[] buffer = new float[img.width * img.height];

  // *************************************
  // Implement here the double convolution
  // *************************************

  for (int y = 1; y < img.height - 1; y++) { // Skip top and bottom edges
    for (int x = 1; x < img.width - 1; x++) { // Skip left and right
      int val=(int) ((buffer[y * img.width + x] / max)*255);
      result.pixels[y * img.width + x]=color(val);
    }
  }
  return result;
}
```

Implement the algorithm, apply it to the image *board1.jpg* and check your results by comparing it with *board1Scharr.bmp*.

As you can see, many edges are detected, which will cause issues when trying to detect the corners of the board. The last part of this exercise session aims at combining color thresholding, blurring and edge detection to cleanly isolate the shape of the board.

Part III

Effective Detection of the Board

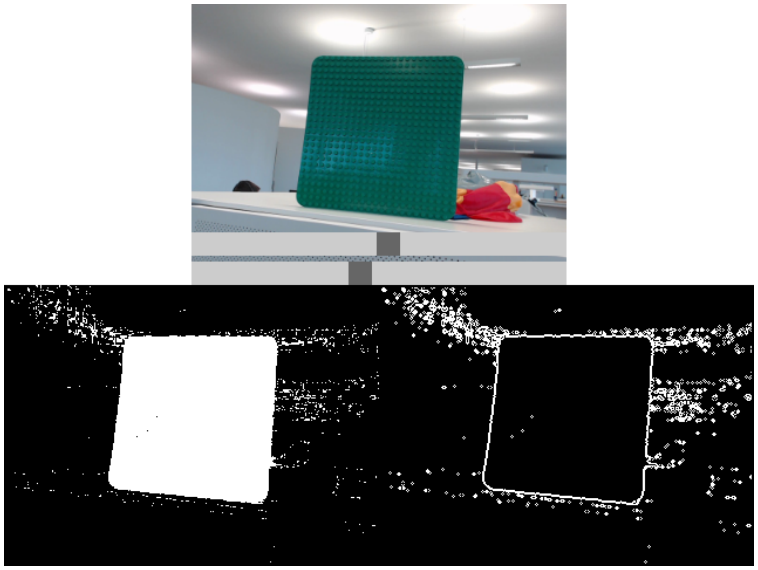
Modify your application to have the following pipeline:

↓ color thresholding

↓ edge detection

↓ thresholding to keep only pixel with values>I, ex. I=100

Tune the parameters of thresholding to remove as much as possible the background of the image of the board (on the picture below, bottom left: hue thresholding, bottom right: edge detection on the thresholded image):



Since the hue of a colour is ill-defined for white and black, `hue()` returns values that are noisy in bright and dark zones: it could "interpret" white as a very light kind of green. Improve the filtering by removing these bright and dark pixels.

Try to insert a Gaussian blur into the pipe-line. Does it improve the edge detection or not?