Introduction to Visual Computing

Assignment# 8

Basics of Image Processing

April 19, 2016

Description

The next step of your project is to control the board of the game with a tangible. To this end, we need to detect a physical object (the Lego green board that you received) 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

- Fetch on Moodle the four pictures of the Lego board that we will use today.

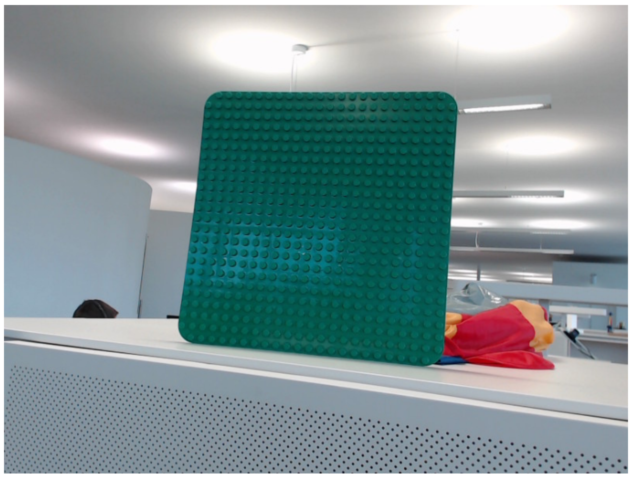
Part I

Colour segmentation

Step 1 – Display an image

Processing has direct support for images. Create a 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)

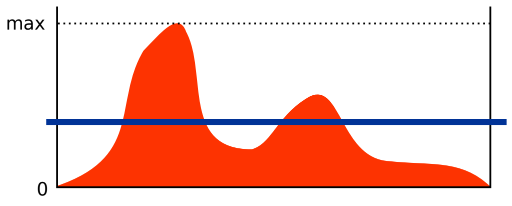
```
PImage img;

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

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

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



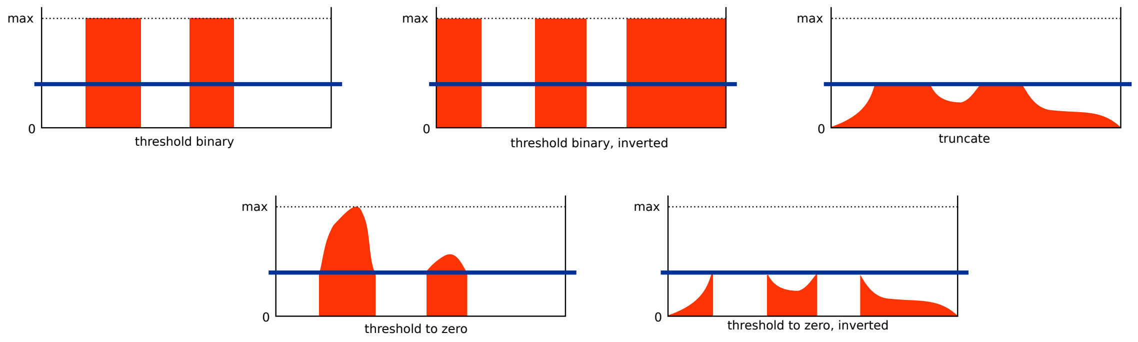
Step 2 – Basic manipulation of the image: thresholding

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

If the following figure represents for instance an intensity profile, with pixel values ranging from 0 to a maximum value, and the blue line represents a fixed threshold:



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



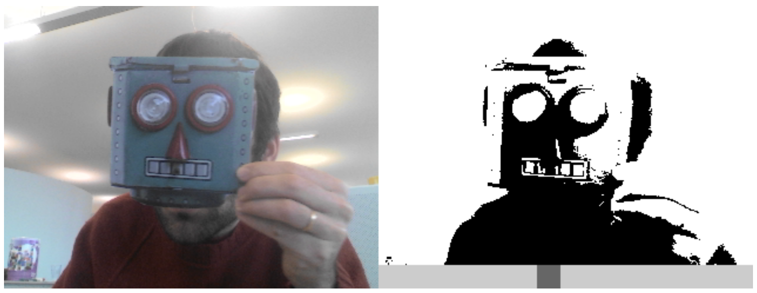
Implement and apply the two first ones on your image, filtering pixels by their intensity with a fixed threshold of 128. Do so by iterating over all pixels:

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

Use the `brightness()` function to obtain one pixel's intensity, and check the documentation of the `color()` function to see how to set pixels' values.

Step 3 – A bit of user interface

Use the `HScrollbar` class that you saw during week 6 to add a scrollbar to the application that sets the threshold level:



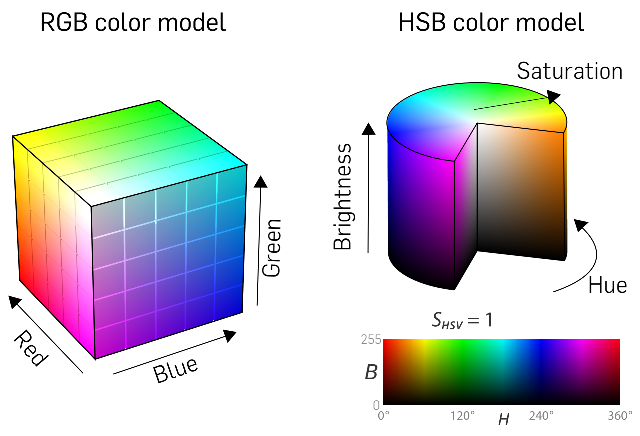
The following code sample shows how to use the scrollbar:

```
HScrollbar thresholdBar;

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

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

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

void draw() {

  background(color(0,0,0));

  //...

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

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

}
```

Step 4 – Colour thresholding

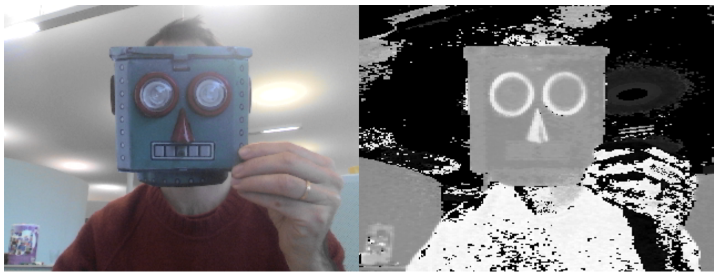
Many image processing algorithms need to select a specific colour in the image. Doing so in the Red-Green-Blue colour space is difficult because it does not effectivelly separate the **hue** of a colour from its brightness and saturation (how dull or vivid a colour looks).

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

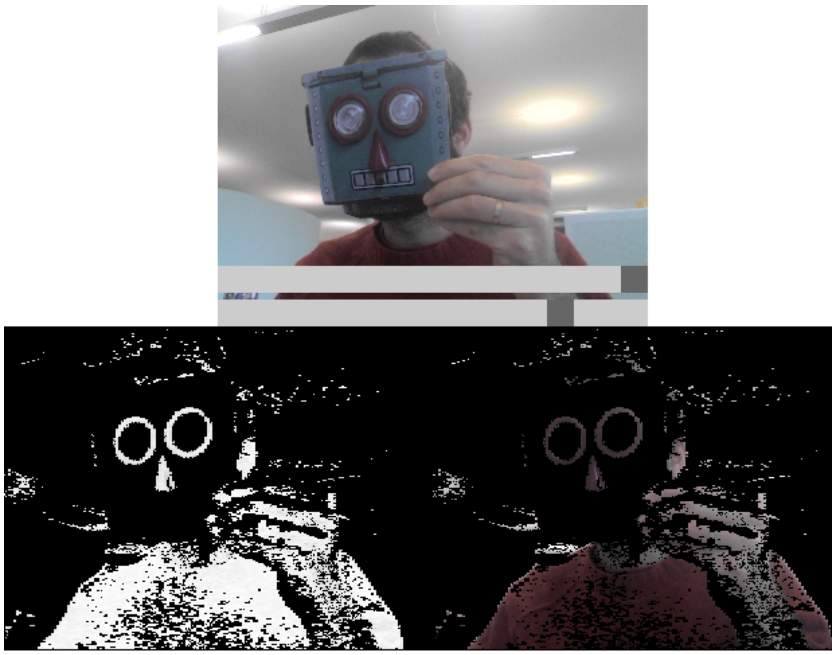


Hue is often expressed as an angle in degree. Processing however represents it as a value between 0 and 255 (likewise saturation and brightness).

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



- Then, modify your application to select only pixels of a specific range of hues with two scrollbars. For instance, on the sample below, only the red zones of the image have been selected:



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, for instance) can be achieved.

In this second part, you will implement a kernel convolution first to blur the image, then to detect edges using the Sobel algorithm.

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 by using the kernel as the matrix of the *weights* of surrounding pixels.

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 weight = 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 the weight
    //     - set result.pixels[y * img.width + x] to this value

    return result;
  }
```

Test your code with the kernels $kernel1$ and $kernel2$ and make sure you understand the role of `weight` (do not hesitate to change it).

Step 2 – Gaussian blur

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

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

Use this kernel to perform a Gaussian blur on the test image.

Step 3 – The Sobel algorithm

The **Sobel algorithm** is the most basic edge detection algorithm. It relies on the combination of two convolutions.

First, apply kernels $hKernel$ and $vKernel$ to your image (using `weight=1.f`), and identify the (symetrical) behaviour of these kernels:

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

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

The Sobel algorithm works as follow:

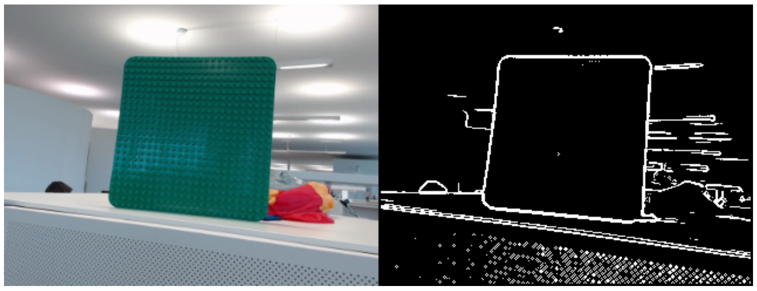
For each (x, y) pair:

- 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 euclidian distance `sum=sqrt(pow(sum_h, 2) + pow(sum_v, 2))`.
- 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 pixels whose values are above a given percentage of the maximum value (you can start with 30%):

```
PImage sobel(PImage img) {

    float[][] hKernel = { { 0,  1, 0  },
                          { 0,  0, 0 },
                          { 0, -1, 0 } };

    float[][] vKernel = { { 0,  0,  0  },
                          { 1,  0, -1 },
                          { 0,  0,  0  } };

    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 = 2; y < img.height - 2; y++) { // Skip top and bottom edges
      for (int x = 2; x < img.width - 2; x++) { // Skip left and right

        if (buffer[y * img.width + x] > (int)(max * 0.3f)) { // 30% of the max
          result.pixels[y * img.width + x] = color(255);
        } else {
          result.pixels[y * img.width + x] = color(0);
        }
      }
    }
    return result;
  }
```

Implement the algorithm and apply it to the still image of the green board:

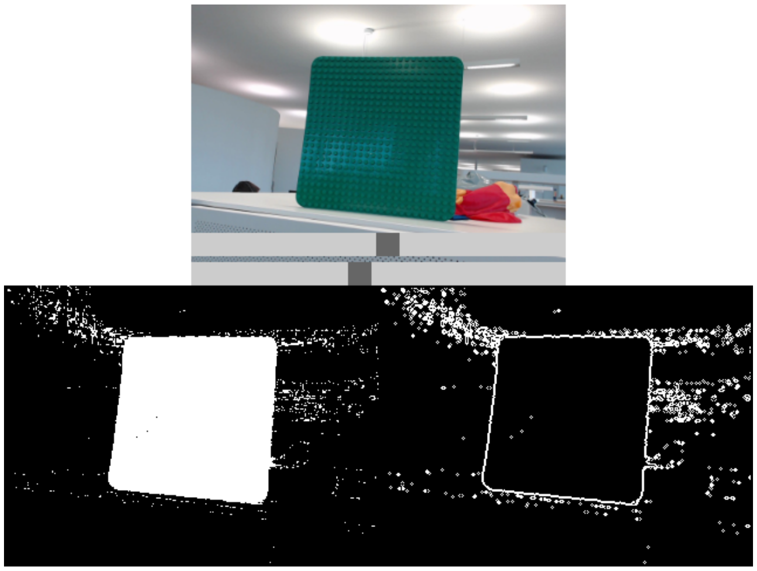


As you can see, many edges are detected, which will cause issues when trying to detect the corners of the board. The last part of the exercise session aims at combining color thresholding, blurring and edge detection to cleanly isolate the shape of the board.

Part III

Effective Detection of the Board

Modify your application to combine hue thresholding and edge detection to remove as much as possible the background of the still 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.

Taking it further (optional)

If you refer back to the HSB color space "cylinder" page 4, you can observe that the hue is also ill-defined for colours whose saturation is close to zero. You can hence improve the filtering robustness by thresholding your image on the three hue, brightness and saturation channels.

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