Report lab3 Computer Vision - Matteo De Gobbi

Task 1

We just call imread, and imshow like in the last lab.

Task 2

As we can see from the opencv headers MouseCallback is defined as:

```
MouseCallback onMouse (aka void (*)(int, int, int, int, void *))
```

A function pointer to a function of void type that takes as input 4 ints and a void pointer.
The first parameter takes value with the type of mouse action that triggered the callback, in the case of left click it's EVENT_LBUTTONDOWN.
The second and third parameters are the coordinates of the pixel on which the user clicked.
The fourth parameter is used as a flag. I didn't use it in this program.
The void* is used as a means to pass variables from the scope on which the callback is defined to the callback function. A void* is used because it allows us to pass a pointer to any type (we just need to cast it back to the correct type in the callback).

```
void pixel_callback(int event, int x, int y, int flags, void *userdata) {
  using namespace cv;
  using namespace std;
  Mat *img_ptr = static_cast<Mat *>(userdata);
  if (event == EVENT_LBUTTONDOWN) {
    cout << img_ptr->at<Vec3b>(y, x) << endl;
  }
}
```

This is the function to execute when the pixel is clicked for task 2:
We cast the void* userdata to a Mat* and then print the value of the pixel clicked using x,y parameters.
To set the callback we just call the setMouseCallback function.

```
// NOTE task 2
cv::imshow("input", in_img);
cv::setMouseCallback("input", &pixel_callback, &in_img);
```

Task 3

This task is very similar to task2 but we compute the color as an average on the 9x9 surrounding pixels to the one clicked, we just loop on the surrounding pixels and take the sum of the pixel for each channel and at the end we divide by the number of pixels summed. We have to be careful because the surrounding pixels may go outside the bounds of the image, I just use a if condition to check and in that case we will compute the average only on the pixels that aren't out of bounds.

```
if (event == EVENT_LBUTTONDOWN) {
  int counter = 0;
  int b_sum = 0;
  int g_sum = 0;
  int r_sum = 0;
  for (int i = -kernel_size / 2; i ≤ kernel_size / 2; i++) {
    for (int j = -kernel_size / 2; j ≤ kernel_size / 2; j++) {
      if (y + i < 0 || x + j < 0 || y + i ≥ in_img.rows ||
          x + j ≥ in_img.cols) {
        continue;
      }
      Vec3b current = in_img.at<Vec3b>(y + i, x + j);
      b_sum += current[0];
      g_sum += current[1];
      r_sum += current[2];
      counter++;
    }
  }
  *ref_color = (Vec3b){(uchar)(b_sum / counter), (uchar)(g_sum / counter),
                       (uchar)(r_sum / counter)};
  cout << *ref_color << endl; // task 3
```

Task 4

For this task (and the next two) we need to be able to pass more than one parameter to the callback function. In order to use the void* parameter in the callback to pass more than one variable I defined a struct type userdata_type:

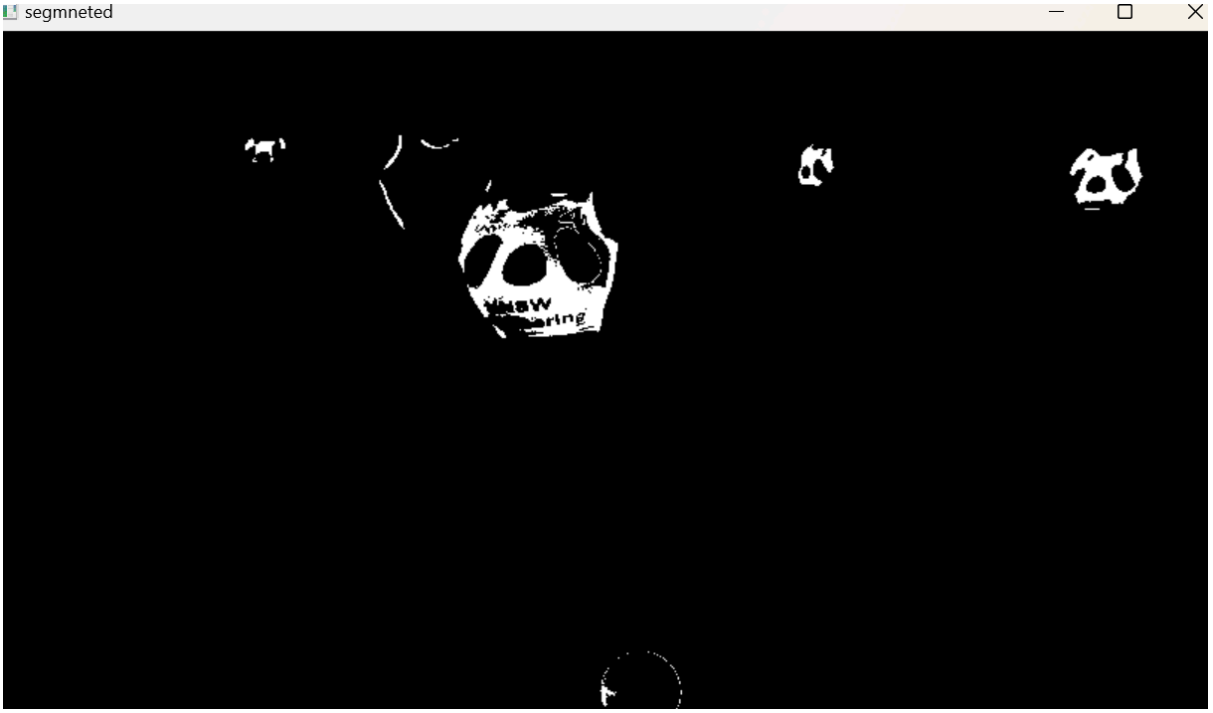
```
struct userdata_type {
   cv::Mat *in;
   cv::Mat *out;
   int threshold;
   cv::Vec3b *reference_color;
   bool task6 = false;
};
```

Which contains two Mat* corresponding to the input and output image, a int that is used as a threshold (for task 4,5,6) a Vec3b* to obtain and use the reference color to segment the robot t-shirts (in tasks 4,5,6) and a boolean task6 that is used as a flag in order to color the shirts of the robots instead of generating only the mask (used in task 6).

After executing the code above in task 3 to compute the reference_color in the surrounding 9x9 pixels to the one clicked we can scan the whole image and compute if the distance of each pixel to the reference_color is bigger or smaller to the threshold provided and act accordingly, either setting the pixel in the output image (the mask) to white if the distance is less than the threshold or to black if the distance is greater.
I decided to use the L1 distance between the colors instead of the euclidean distance because it seemed to work better in practice, as a threshold I found experimentally that 100 worked well.

```
void segment_shirt(const cv::Mat &in_img, cv::Mat &out_img, cv::Vec3b ref_color,
                   int threshold) {
  using namespace cv;
  using namespace std;
  cout << ref_color << endl;
  for (int r = 0; r < out_img.rows; r++) {
    for (int c = 0; c < out_img.cols; c++) {
      Vec3b curr_color = in_img.at<Vec3b>(r, c);
      int distance = abs(curr_color[0] - ref_color[0]) +
                     abs(curr_color[1] - ref_color[1]) +
                     abs(curr_color[2] - ref_color[2]);
      // float distance = cv::norm(curr_color, ref_color);
      if (distance > threshold) {
        out_img.at<Vec3b>(r, c) = {0, 0, 0};
      } else {
        out_img.at<Vec3b>(r, c) = {255, 255, 255};
      }
    }
  }
```

The output mask when clicking on the shirt of the middle robot is this, we can see how all the yellow shirts are visible but we can also see the outline of the ball in the bottom:

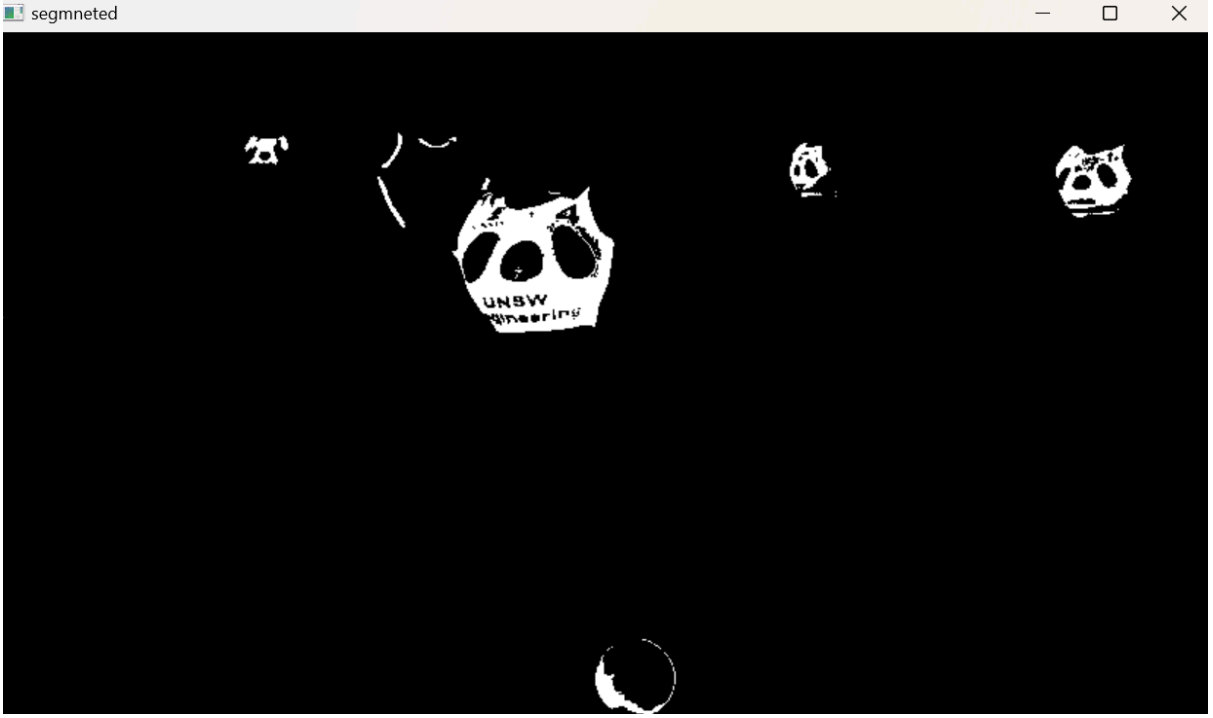


Task 5

To convert the image to HSV we just need to call this function:

```
cv::cvtColor(in_img, in_img_hsv, cv::COLOR_BGR2HSV);
```

Which converts the image from BGR to HSV the code for the callback is the same as task 4. And these are the results:



We can see how the shirts are more visible but also how the ball has a bigger outline in the HSV version, if we lower the threshold we get similar results to the BGR version.

Task 6

In this task instead of just generating the mask as in task 4 and 5 we also need to use it to generate an image where the pixels corresponding to where the mask is white are colored in the color provided (92, 37, 201) while the other pixels are the same as in the original image. We just execute this code after having generated the mask (stored in the variable out_img) as in task 4 which sets the pixel color as the one in the original image or to select_color depending on the value of the mask.

```
const Vec3b select_color = {92, 37, 201};
for (int r = 0; r < out_img.rows; r++) {
  for (int c = 0; c < out_img.cols; c++) {
    Vec3b curr_color = out_img.at<Vec3b>(r, c);
    if (curr_color[0] == 0 && curr_color[1] == 0 && curr_color[2] == 0) {
      out_img.at<Vec3b>(r, c) = in_img.at<Vec3b>(r, c);
    } else {
      out_img.at<Vec3b>(r, c) = select_color;
    }
  }
}
cv::imshow("selected task6", out_img);
cv::waitKey(1);
```

Note on updating the same window on click: in order to redraw the image in the same window we need to call cv::waitKey(1) instead of cv::waitKey(0) so it waits for 1ms and the program has time to process the draw request from the previous imshow call.