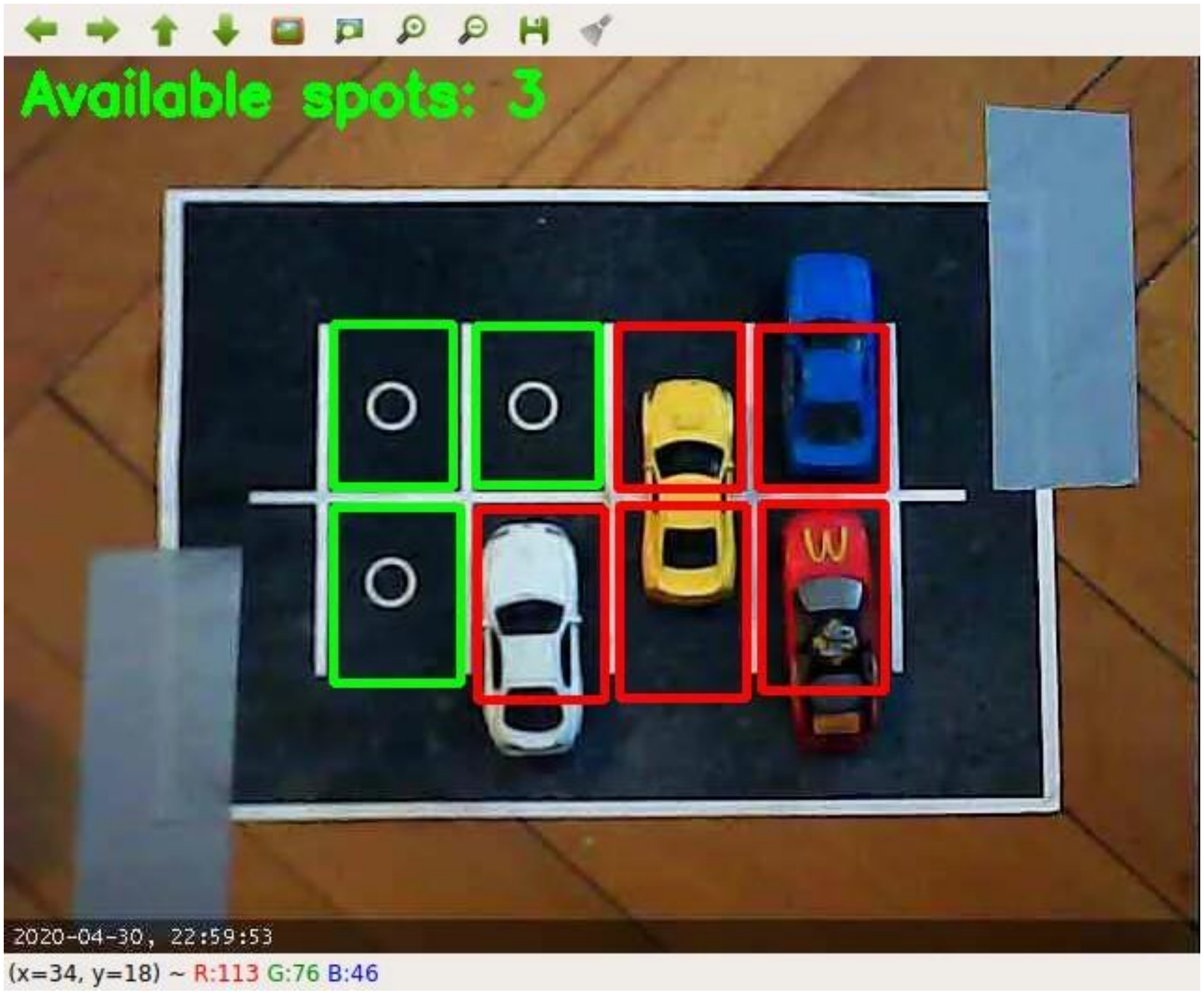
AL ENABLED CAR PARKING USING OPENCV

TEAM ID: NM2023TMID18442

PROJECT NAME:AL ENABLED CAR PARKING USING OPENCV



Backstory

One of the most annoying things happening in our era is looking for a parking spot and not being able to find one, especially when you had to be somewhere five minutes ago and you are currently looking for a parking spot for 20 minutes now.

However, any problem must have a solution, or more solutions, based on complexity and efficiency. There are many solutions for smart parking systems out there, including deep learning implementations, weight sensors, light sensors and all of that science fiction stuff that surrounds the world these days.

This article is focusing on guiding you trough one of the simplest approaches to a smart parking system using only a webcam and few lines of code.

Overview

The concept behind this solution is quite simple. It is composed of two scripts with the following roles:

1. Select the coordinates of the parking spaces and save them into a file.
2. Get the coordinates from the file and decide if the spot is available or not.

The reason for splitting this solution in two scripts is strictly related to avoiding to select the spots every time you want to see if there are any available spots especially if used the same location as before.

To keep this as simple as possible, from now on I will refer to these scripts as selector and detector.

Dependencies

First of all you will need to have python installed. To do so, you can visit https://www.python.org/downloads/ to download and install python. For this article I used python 3.7.6 but other versions like 3.6 or 3.8 are working as well. To check your python version you can write in your console `python --version` and it will return your installed python version.

```
C:\Users\Razvan>python --version
Python 3.7.6
```

Before starting to build the project dependencies, I strongly advise you to set up a virtual environment. You can read more about virtual environments at https://docs.python.org/3.7/tutorial/venv.html.

Also you can use conda to create and manage environments. See https://docs.anaconda.com/anaconda/ for more info.

After everything is set up in python, the most important dependency would be OpenCV library. To add it to your virtual environment through pip you can run `pip install opencv-python`.

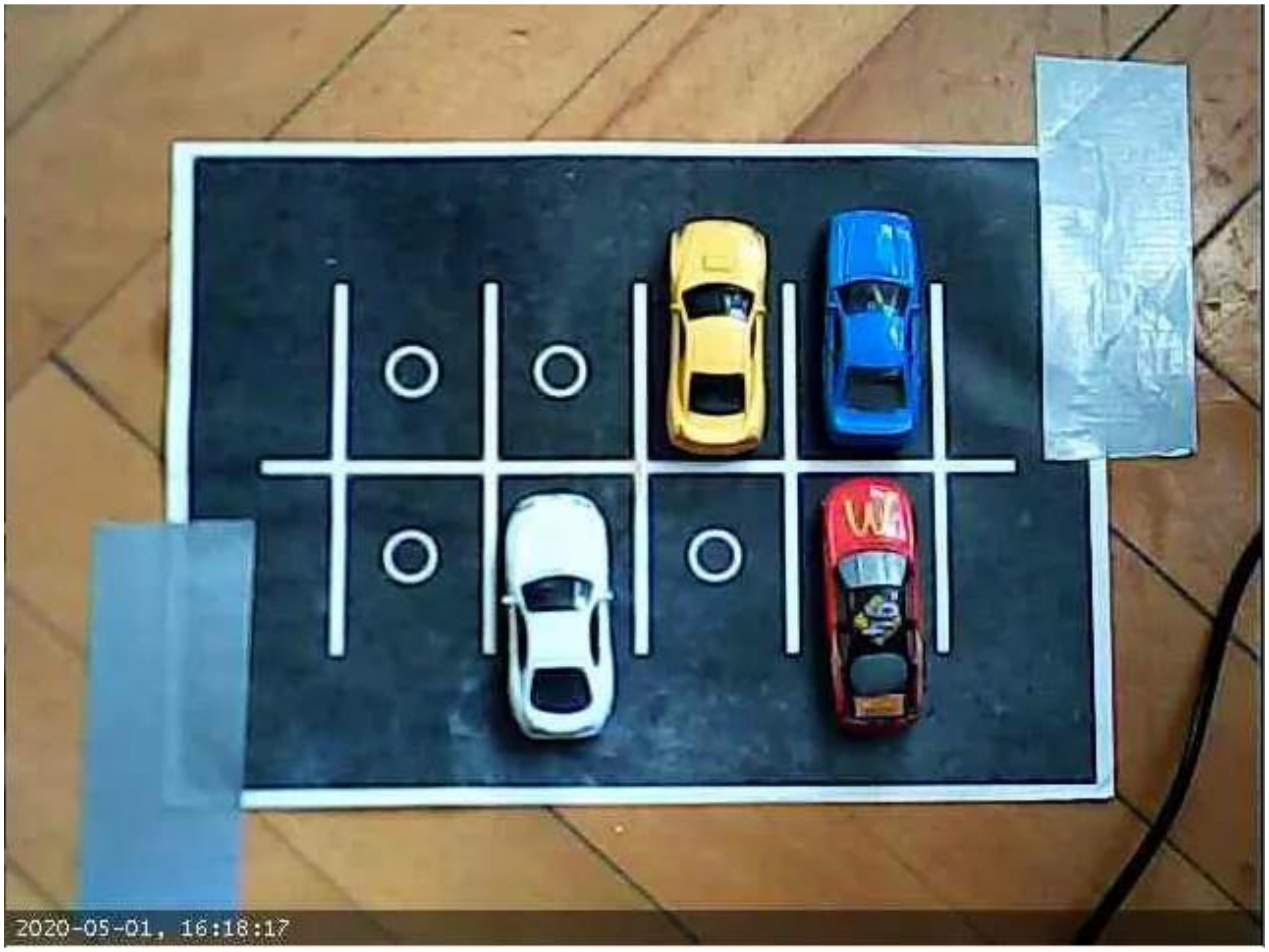
To check if everything is set up correctly we can print the current OpenCV version available on the environment using the `cv2.__version__`command.

```
(OpenCV) C:\Users\Razvan>python
Python 3.7.6 (default, Jan  8 2020, 20:23:39) [MSC v.1916 64 bit
(AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import cv2
>>> print(cv2.__version__)
4.2.0
>>>
```

Also in the first line you can see that I am already using a virtual environment called OpenCV for this project.

Time to take action

First of all we will need to setup a parking lot camera. In my case, as I don't have any parking lots that I can see from my windows I choose to use my old car toys and a printed paper to play with. Also I set up a webcam right above the *parking lot* to get a good image, so the image we are working on looks like this:



The selector

Now, let's get to the coding part. First we need to build the selector. We start with importing the modules we need (this is available for both selector and detector)

```
import cv2
import csv
```

After that we can start working on getting the image on which we will select the parking spots. For that we can take the first frame provided by the webcam, save it and use the picture to select the spots. The following code works like this:

1. Opens the video stream in `image` variable; `suc` determines if the stream was opened successfully.
2. Writes the first frame into frame0.jpg.
3. The stream is released and all windows are closed.
4. The new saved picture is read in `img` variable.

```
VIDEO_SOURCE = 1

cap = cv2.VideoCapture(VIDEO_SOURCE)
suc, image = cap.read()

cv2.imwrite("frame0.jpg", image)

cap.release()
cv2.destroyAllWindows()

img = cv2.imread("frame0.jpg")
```

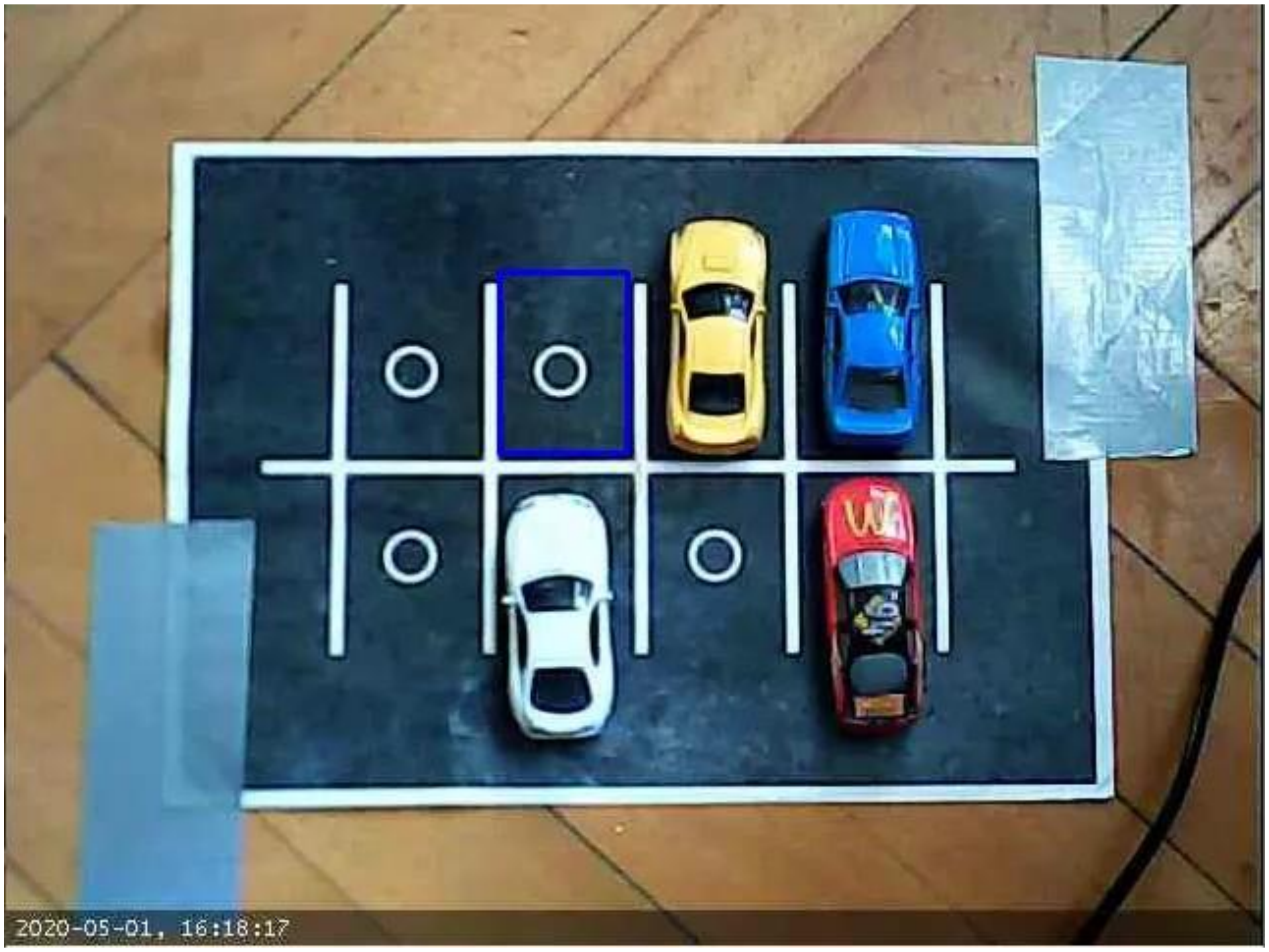
Now that we have saved the first frame and opened it in the `img` variable we can use selectROIs function to mark our parking spots. ROIs are defined as regions of interest and represents a portion of the image on which we will apply different functions and filters to get our results.

Back to selectROIs function, this will return a list (type: numpy.ndarray) which contains the numbers we need to assemble images with their boundaries as our ROIs.

```
r = cv2.selectROIs('Selector', img, showCrosshair = False, fromCenter
= False)
```

Our list will be saved in `r` variable. The parameters given to `cv2.selectROIs` function are the following:

1. 'Selector' is the name of the window that will let us select the ROIs.
2. img is the variable containing the image on which we want to select.
3. showCrosshair = Flase removes the center lines inside the selection. This can be set to True as there's no impact on the result.
4. fromCenter = False is quite an important parameter, as if this would have been set to True the proper selection would have been much harder.



After selecting all of the parking spots, it's time to write them into a .csv file. To do that, first, we need to transform our `r` variable into a python list. For this we can use `rlist = r.tolist()` command.

After we have our proper data, we can save it into the .csv file we will use further.

```
with open('data/rois.csv', 'w', newline='') as outf:
  csvw = csv.writer(outf)
  csvw.writerows(rlist)
```

The detector

Now that we're done with selecting parking spots, it's time to do some image processing. The way I approached this was:

1. Get coordinates from the .csv file.
2. Build a new image out of it.
3. Apply Canny function available in OpenCV.
4. Count the white pixels inside the new image.
5. Establish a pixel range within the spot would be occupied.
6. Draw a red or green rectangle on the live feed.

For all of these operations we need to define a function to be applied for each spot. This is how the function looks like:

```
def drawRectangle(img, a, b, c, d):
    sub_img = img[b:b + d, a:a + c]

    edges = cv2.Canny(sub_img, lowThreshold, highThreshold)

    pix = cv2.countNonZero(edges)

    if pix in range(min, max):
        cv2.rectangle(img, (a, b), (a + c, b + d), (0, 255, 0), 3)
        spots.loc += 1
    else:
        cv2.rectangle(img, (a, b), (a + c, b + d), (0, 0, 255), 3)
```

Now that we have the function we need, all we have to do is to apply it to every set of coordinates in the .csv file, right? Could be, but we can make it even better.

First of all we have some parameters which would be nice if we could adjust them in real time while the script is running, also via a GUI. To do that we need to build a few track bars. Fortunately, OpenCV gives us the opportunity to do that without extra libraries.

First we need a callback function which does nothing but is required in as a parameter for creating track bars with OpenCV. Actually the callback parameter has a well defined purpose but we don't use it here. To read more about this visit the OpenCV docs.

```
def callback(foo):
    pass
```

Now we need to create the track bars. To do so we will use `cv2.namedWindow` and `cv2.createTrackbar` functions.

```
cv2.namedWindow('parameters')
cv2.createTrackbar('Threshold1', 'parameters', 186, 700, callback)
cv2.createTrackbar('Threshold2', 'parameters', 122, 700, callback)
cv2.createTrackbar('Min pixels', 'parameters', 100, 1500, callback)
cv2.createTrackbar('Max pixels', 'parameters', 323, 1500, callback)
```

Now that we created the GUI to operate the parameters there is one thing left. This would be the number of available spots in the image. Which is defined in the drawRectangle as spots.loc. This is a static variable which must be defined at the beginning of our program. The reason why this variable is static is because we want every drawRectangle function we call to write it's result in the same global variable, and not a separate variable for each function. This prevents the returned number of available spaces to be higher than the actual number of available spaces.

To implement this we just have to create the spots class with it's loc static variable.

```
class spots:
    loc = 0
```

Now that we’re almost ready, we just need to get the data from the .csv file, convert all its data into integers and apply our built functions in an infinite loop.

```
with open('data/rois.csv', 'r', newline='') as inf:
    csvr = csv.reader(inf)
    rois = list(csvr)

rois = [[int(float(j)) for j in i] for i in rois]

VIDEO_SOURCE = 1
cap = cv2.VideoCapture(VIDEO_SOURCE)

while True:
    spots.loc = 0

    ret, frame = cap.read()
    ret2, frame2 = cap.read()

    min = cv2.getTrackbarPos('Min pixels', 'parameters')
    max = cv2.getTrackbarPos('Max pixels', 'parameters')
    lowThreshold = cv2.getTrackbarPos('Threshold1', 'parameters')
    highThreshold = cv2.getTrackbarPos('Threshold2', 'parameters')

    for i in range(len(rois)):
        drawRectangle(frame, rois[i][0], rois[i][1], rois[i][2],
rois[i][3])

    font = cv2.FONT_HERSHEY_SIMPLEX
    cv2.putText(frame, 'Available spots: ' + str(spots.loc), (10, 30),
font, 1, (0, 255, 0), 3)
    cv2.imshow('Detector', frame)

    canny = cv2.Canny(frame2, lowThreshold, highThreshold)
    cv2.imshow('canny', canny)

    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

cap.release()
cv2.destroyAllWindows()
```

To infinity and beyond

What is happening inside our loop it’s actually a bit more complicated that just calling our build functions.

First we initialize the number of available spaces to 0 to prevent the addition of numbers for every frame.

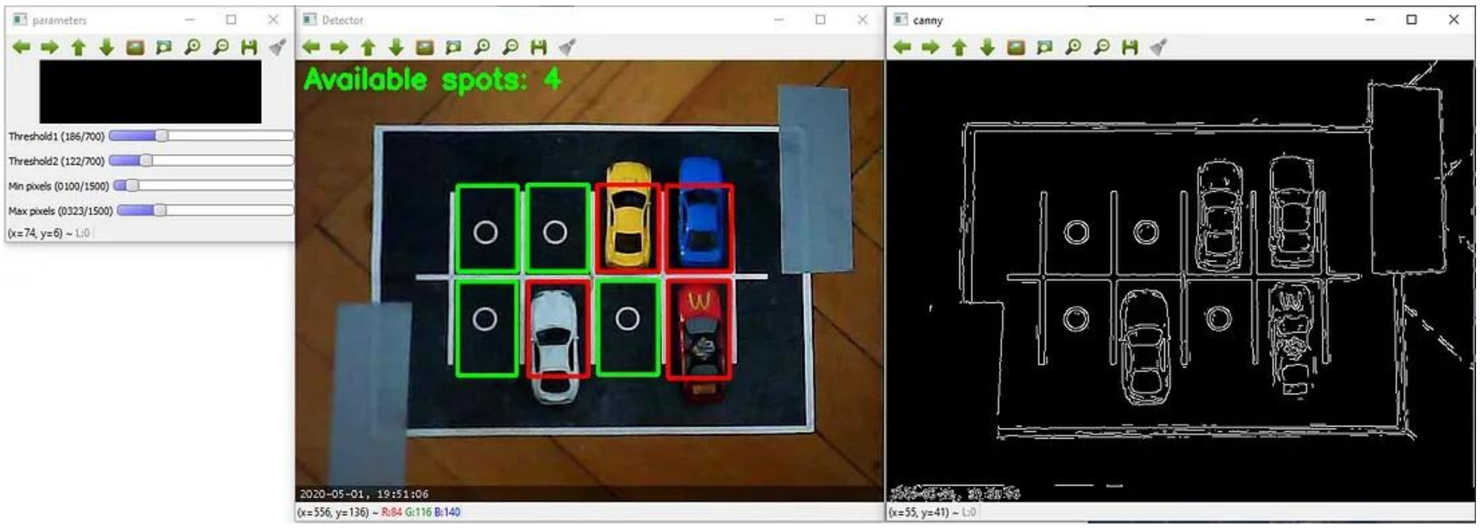
Then we start two streams to show the real image and the image that is processed. This helps for a better understanding of how this script works and how the image is processed.

After that we need to get the values of the parameters set in the parameters GUI we created, each iteration. This is done with the `cv2.getTrackbarPos` function.

Now the most important part takes places, the appliance of drawRectangle function to all the coordinates taken by Selector script.

What’s left to do now is to write the number of available spots on the resulted image, display the Canny function result and, obviously, a well known way to stop our loop.

Voila! we have a smart parking project now!



Threshold1 (186/700)
Threshold2 (122/700)
Min pixels (0100/1500)
Max pixels (0323/1500)
(x=74, y=6) ~ L:0
(x=556, y=136) ~ R:84 G:116 B:140
(x=55, y=41) ~ L:0