# Приложение

## Файл App.py

import configparser

from VideoScanner import VideoScanner

class App:

def \_\_init\_\_(self):

self.config = configparser.ConfigParser()

self.config.read("config.ini")

self.\_video = VideoScanner(self.config)

self.data = {}

def run(self):

self.\_video.set()

self.data = self.\_video.scan()

self.export()

def export(self):

exportFormat = self.config['Export']['exportFormat']

if exportFormat == 'RawTXT':

self.ExportAsRawTXT()

elif exportFormat == 'PythonList':

self.ExportAsPythonList()

elif exportFormat == 'PythonDict':

self.ExportAsPythonDict()

elif exportFormat == 'JSON':

self.ExportAsJSON()

elif exportFormat == 'NumpyArray':

self.ExportAsNumpyArray()

elif exportFormat == 'Excel':

self.ExportAsExcel()

elif exportFormat == 'Graph':

self.ExportAsGraph()

def ExportAsRawTXT(self):

with open(self.config['Export']['exportFileName']+'.txt', 'w', encoding='utf-8') as file:

file.write('\n'.join(map(str, self.data.values())))

def ExportAsPythonList(self):

with open(self.config['Export']['exportFileName']+'.txt', 'w', encoding='utf-8') as file:

file.write(str(list(self.data.values())))

def ExportAsPythonDict(self):

with open(self.config['Export']['exportFileName']+'.txt', 'w', encoding='utf-8') as file:

file.write(str(self.data))

def ExportAsJSON(self):

import json

with open(self.config['Export']['exportFileName']+'.json', 'w', encoding='utf-8') as file:

file.write(json.dumps(self.data))

def ExportAsNumpyArray(self):

import numpy as np

np.save(self.config['Export']['exportFileName'], np.array(list(self.data.values())))

def ExportAsExcel(self):

import xlsxwriter

workbook = xlsxwriter.Workbook(self.config['Export']['exportFileName']+'.xlsx')

worksheet = workbook.add\_worksheet()

worksheet.write(0, 1, 'Секунда')

worksheet.write(0, 2, 'Значение')

for i, sec in enumerate(self.data):

worksheet.write(i+1, 1, sec)

worksheet.write(i+1, 2, self.data[sec])

workbook.close()

def ExportAsGraph(self):

import matplotlib.pyplot as plt

names = list(self.data.keys())

values = list(self.data.values())

fig, ax = plt.subplots()

ax.plot(names, values)

plt.show()

## VideoScanner.py

from enum import Enum, auto

import cv2

import numpy as np

class SetterState(Enum):

Transforming = auto()

Placement = auto()

Naming = auto()

Scanning = auto()

Fixing = auto()

class VideoScanner:

def \_\_init\_\_(self, config):

self.config = config

self.path = config['Video']['videoPath']

self.\_capture = cv2.VideoCapture(self.path)

self.fps = self.\_capture.get(5)

self.cropping = None

self.croppingHistory = []

self.croppingArea = [(), ()]

self.state = SetterState.Transforming

self.scaleF = 1

self.rotate = 0

self.digits = []

self.noNamedSegments = []

self.segmentsHistory = []

self.nameHistory = []

self.name\_index = 0

self.noNamedDigits = []

self.error\_count = 0

self.selection = []

self.decimalPoint = int(self.config['Video']['decimalPoint'])

self.totalFrameCount = self.\_capture.get(cv2.CAP\_PROP\_FRAME\_COUNT)

self.global\_scan\_data = {}

self.currentSecScan = int(self.config['Video']['startSec'])

self.\_capture.set(1, round(self.fps \* self.currentSecScan, 1))

self.scan\_data = []

\_, self.source\_img = self.\_capture.read()

self.frame = self.source\_img.copy()

self.sizeY, self.sizeX, \_ = self.frame.shape

self.ratio = self.sizeY / self.sizeX

def set(self):

self.showFrame()

cv2.setMouseCallback('Frame', self.onClick)

self.transform()

self.placement()

self.naming()

def \_scale(self):

self.sizeY, self.sizeX, \_ = self.frame.shape

self.ratio = self.sizeY / self.sizeX

if self.sizeX > 900 or self.sizeY > 900:

self.frame = cv2.resize(self.frame, (round(900 / self.ratio), 900))

self.scaleF = 900 / self.sizeY

elif self.sizeX < 600 or self.sizeY < 600:

self.frame = cv2.resize(self.frame, (round(600 / self.ratio), 600))

self.scaleF = 600 / self.sizeY

else:

self.scaleF = 1

def \_rotate(self):

self.frame = np.ascontiguousarray(np.rot90(self.frame, self.rotate), dtype=np.uint8)

def \_drawSegments(self):

[seg.draw(self.frame) for seg in self.segmentsHistory]

def \_drawPreview(self):

if not self.scan\_data:

return

block = round(self.frame.shape[0] / 9)

digit\_width = block \* 5

digit\_display\_image = np.zeros(

(self.frame.shape[0], round(1.2 \* digit\_width \* len(self.digits) + 1 \* block), 3), np.uint8)

self.previewSize = (self.frame.shape[0], digit\_width \* len(self.digits))

segments\_positions = [

((1 \* block, 0 \* block), (4 \* block, 1 \* block)),

((0 \* block, 1 \* block), (1 \* block, 4 \* block)),

((4 \* block, 1 \* block), (5 \* block, 4 \* block)),

((1 \* block, 4 \* block), (4 \* block, 5 \* block)),

((0 \* block, 5 \* block), (1 \* block, 8 \* block)),

((4 \* block, 5 \* block), (5 \* block, 8 \* block)),

((1 \* block, 8 \* block), (4 \* block, 9 \* block))

]

for i, d in enumerate(self.digits):

main\_anchor = np.array([round(digit\_width \* i \* 1.2), 0])

for s in range(7):

if list(self.scan\_data[i].values())[s]:

cv2.rectangle(digit\_display\_image,

main\_anchor + segments\_positions[s][0], main\_anchor + segments\_positions[s][1],

(255, 255, 255), -1)

else:

cv2.rectangle(digit\_display\_image,

main\_anchor + segments\_positions[s][0], main\_anchor + segments\_positions[s][1],

(50, 50, 50), -1)

anchor = np.array([round(len(self.digits) \* digit\_width \* 1.2), 0])

height = round(9 \* block \* self.fps \* self.currentSecScan / self.totalFrameCount)

cv2.rectangle(digit\_display\_image, anchor, anchor + (block // 2, height), (0, 255, 0), -1)

cv2.rectangle(digit\_display\_image, anchor + (block // 2, 0),

anchor + (block, self.frame.shape[0] // 2), Segment.offColor, -1)

cv2.rectangle(digit\_display\_image, anchor + (block // 2, self.frame.shape[0] // 2), anchor + (block, self.frame.shape[0]),

Segment.onColor, -1)

self.frame = np.concatenate((self.frame, digit\_display\_image), axis=1, dtype=np.uint8)

def \_scan(self, nextFrame=True):

self.\_capture.set(1, round(self.fps \* self.currentSecScan, 1))

ret, self.source\_img = self.\_capture.read()

if ret:

if nextFrame:

self.currentSecScan += 1

self.scan\_data = []

scan\_interrupt = []

for d in self.digits:

scan = d.scan(self.source\_img)

scan\_interrupt.append(scan[0])

self.scan\_data.append(scan[1])

self.error\_count = 0

for i, res in enumerate(scan\_interrupt):

if not res[0]:

self.digits[i].is\_broken = True

self.error\_count += 0

if nextFrame:

return int(''.join([str(i[1]) for i in scan\_interrupt])) / (10 \*\* self.decimalPoint)

def transform(self):

self.state = SetterState.Transforming

while True:

self.showFrame()

cv2.setWindowTitle('Frame', 'Transforming')

key = cv2.waitKey()

if key == 13:

break

elif key == 8:

if self.croppingHistory:

self.croppingHistory.pop(-1)

self.cropping = self.croppingHistory[-1] if self.croppingHistory else None

self.showFrame()

elif ord('r') == key:

self.rotate = (self.rotate + 1) % 4

elif key == -1:

quit()

else:

print(key)

def placement(self):

self.state = SetterState.Placement

while True:

self.showFrame()

cv2.setWindowTitle('Frame', 'Placement')

key = cv2.waitKey()

if key == 13:

if not (len(self.segmentsHistory) % 7) and self.segmentsHistory:

break

else:

cv2.setWindowTitle('Frame', 'Placement (Miss much segments count)')

cv2.waitKey(1000)

elif key == -1:

quit()

elif key == 8:

self.removeLast()

else:

print(key)

def naming(self):

self.state = SetterState.Naming

for i in range(len(self.noNamedSegments) // 7):

self.noNamedDigits.append(Digit(self))

self.showFrame()

cv2.setWindowTitle('Frame', 'Naming')

while True:

key = cv2.waitKey()

if key == 8:

if self.nameHistory:

seg = self.nameHistory.pop(-1)

if len(seg.digit.segments) == 7:

d = seg.digit

self.digits.remove(d)

self.noNamedDigits.insert(0, d)

self.noNamedSegments.append(seg)

seg.removeName()

self.name\_index -= 1

self.showFrame()

elif key == 13 and self.allNamed():

break

elif key == -1:

quit()

def scan(self):

cv2.setWindowTitle('Frame', 'Scanning')

self.state = SetterState.Scanning

[dig.sort() for dig in self.digits]

while True:

data = self.\_scan(True)

print(f'{self.currentSecScan}-{data}')

self.global\_scan\_data[self.currentSecScan] = data

self.showFrame()

key = cv2.waitKey(10 \*\* self.error\_count)

if key == 102:

self.fixing()

self.state = SetterState.Scanning

if round(self.fps \* (self.currentSecScan + 1), 1) > self.totalFrameCount:

print('Done')

break

return self.global\_scan\_data

def fixing(self):

self.state = SetterState.Fixing

self.selection = []

cv2.setWindowTitle('Frame', 'Fixing')

while True:

self.\_scan(False)

key = cv2.waitKey()

if key == (119, 97, 115, 100)[(0 + self.rotate) % 4]:

[seg.move((0, -2)) for seg in self.selection]

elif key == (119, 97, 115, 100)[(1 + self.rotate) % 4]:

[seg.move((-2, 0)) for seg in self.selection]

elif key == (119, 97, 115, 100)[(2 + self.rotate) % 4]:

[seg.move((0, 2)) for seg in self.selection]

elif key == (119, 97, 115, 100)[(3 + self.rotate) % 4]:

[seg.move((2, 0)) for seg in self.selection]

elif key == 102:

self.selection = self.segmentsHistory.copy()

[s.select() for s in self.selection]

elif key == 122:

if self.croppingArea[0]:

print(type(Segment.offColor))

Segment.offColor = tuple(self.source\_img[self.croppingArea[0][1], self.croppingArea[0][0]])

Segment.offColorSum = np.sum(self.source\_img[self.croppingArea[0][1], self.croppingArea[0][0]])

print(type(Segment.offColor))

elif key == 120:

if self.croppingArea[0]:

Segment.onColor= tuple(self.source\_img[self.croppingArea[0][1], self.croppingArea[0][0]])

Segment.onColorSum = np.sum(self.source\_img[self.croppingArea[0][1], self.croppingArea[0][0]])

elif key == 13:

break

elif ord('r') == key:

self.rotate = (self.rotate + 1) % 4

self.showFrame()

[s.deselect() for s in self.selection]

def showFrame(self):

self.frame = self.source\_img.copy()

self.\_cropping()

self.\_scale()

self.\_rotate()

self.sizeY, self.sizeX, \_ = self.frame.shape

self.\_drawSegments()

self.\_drawPreview()

cv2.imshow('Frame', self.frame)

def \_cropping(self):

if self.cropping is not None:

self.frame = self.frame[self.cropping[0][1]:self.cropping[1][1], self.cropping[0][0]:self.cropping[1][0]]

def convertCords(self, pos):

# Координаты с экрана → Кординаты исходного кадра

frameSize = (self.sizeY, self.sizeX)

if self.rotate == 0:

pos = (pos[0], pos[1])

elif self.rotate == 1:

pos = (frameSize[0] - pos[1], pos[0])

elif self.rotate == 2:

pos = (frameSize[1] - pos[0], frameSize[0] - pos[1])

elif self.rotate == 3:

pos = (pos[1], frameSize[1] - pos[0])

else:

raise IndexError

pos = (round(pos[0] / self.scaleF), round(pos[1] / self.scaleF))

if self.cropping is not None:

pos = (pos[0] + self.cropping[0][0], pos[1] + self.cropping[0][1])

return pos

def showedCords(self, pos):

# Кординаты исходного кадра → Координаты на экране

if self.cropping is not None:

pos = (pos[0] - self.cropping[0][0], pos[1] - self.cropping[0][1])

pos = (round(pos[0] \* self.scaleF), round(pos[1] \* self.scaleF))

frameSize = (self.sizeY, self.sizeX)

if self.rotate == 0:

pos = (pos[0], pos[1])

elif self.rotate == 1:

pos = (pos[1], frameSize[0] - pos[0])

elif self.rotate == 2:

pos = (frameSize[1] - pos[0], frameSize[0] - pos[1])

elif self.rotate == 3:

pos = (frameSize[1] - pos[1], pos[0])

else:

raise IndexError

return pos

def onClick(self, event, posX, posY, flags, param):

pos = self.convertCords((posX, posY))

if self.state == SetterState.Transforming:

if event == 1:

self.croppingArea[0] = pos

elif event == 4:

if self.croppingArea[0] != pos:

self.croppingArea[1] = pos

self.croppingArea = [(min(self.croppingArea[0][0], self.croppingArea[1][0]),

min(self.croppingArea[0][1], self.croppingArea[1][1])),

(max(self.croppingArea[0][0], self.croppingArea[1][0]),

max(self.croppingArea[0][1], self.croppingArea[1][1]))]

if abs(self.croppingArea[0][0] - self.croppingArea[1][0]) + abs(

self.croppingArea[0][1] - self.croppingArea[1][1]) < 50:

cv2.setWindowTitle('Frame', 'Transforming (to small area)')

cv2.waitKey(1000)

cv2.setWindowTitle('Frame', 'Transforming')

else:

self.cropping = tuple(self.croppingArea)

self.croppingHistory.append(tuple(self.croppingArea))

self.croppingArea = [(), ()]

self.showFrame()

elif self.state == SetterState.Placement:

if event == 1:

self.setSegment(pos)

self.showFrame()

elif self.state == SetterState.Naming:

if event == 1:

if self.noNamedSegments:

seg = min(self.noNamedSegments, key=lambda p: (p.pos[0] - pos[0]) \*\* 2 + (p.pos[1] - pos[1]) \*\* 2)

seg.name = SN.getName(self.name\_index)

self.name\_index += 1

self.nameHistory.append(seg)

digit = self.noNamedDigits[0]

seg.setDigit(digit)

self.noNamedSegments.remove(seg)

self.showFrame()

if self.noNamedDigits[0].isNamed():

self.digits.append(self.noNamedDigits.pop(0))

elif self.state == SetterState.Fixing:

if event == 1:

seg = min(self.segmentsHistory,

key=lambda p: (p.pos[0] - pos[0]) \*\* 2 + (p.pos[1] - pos[1]) \*\* 2)

[s.deselect() for s in self.selection]

self.selection = [seg]

seg.select()

self.showFrame()

elif event == 2:

seg = min([s for s in self.segmentsHistory if not s.isSelected],

key=lambda p: (p.pos[0] - pos[0]) \*\* 2 + (p.pos[1] - pos[1]) \*\* 2)

self.selection.append(seg)

seg.select()

self.showFrame()

elif event == 3:

self.croppingArea[0] = pos

def setSegment(self, pos):

new\_seg = Segment(pos, self)

self.noNamedSegments.append(new\_seg)

self.segmentsHistory.append(new\_seg)

def removeLast(self):

if self.segmentsHistory:

seg = self.segmentsHistory[-1]

self.segmentsHistory.remove(seg)

self.noNamedSegments.remove(seg)

self.showFrame()

def allNamed(self):

return not self.noNamedSegments

class Segment:

size = 7

offColorSum = 594

onColorSum = 139

offColor = (195, 208, 190)

onColor = (107, 108, 90)

def \_\_init\_\_(self, position, setter):

self.pos = position

self.videoSetter = setter

self.isSelected = False

self.name = None

self.digit = None

def select(self):

self.isSelected = True

def deselect(self):

self.isSelected = False

def setDigit(self, digit):

self.digit = digit

digit.setSegment(self)

def removeName(self):

self.digit.segments.remove(self)

self.name = None

self.digit = None

def scan(self, frame):

color = np.sum(self.getColor(frame, toList=False))

offDif = abs(color - self.offColorSum)

onDif = abs(color - self.onColorSum)

return onDif < offDif

def getColor(self, frame, pos=None, toList=True):

if pos is None:

# pos = self.pos[1], self.pos[0]

pos = self.pos

return frame[pos[1], pos[0]].tolist() if toList else frame[pos[1], pos[0]]

def draw(self, frame):

pos = self.videoSetter.showedCords(self.pos)

cv2.rectangle(frame,

(pos[0] - self.size, pos[1] - self.size),

(pos[0] + self.size, pos[1] + self.size),

self.getColor(frame, pos),

-1)

color = 0, 0, 0

if self.isSelected:

color = 255, 0, 255

elif self.digit is not None and self.digit.is\_broken:

color = 0, 255, 255

elif self.digit is not None and self.digit.isNamed():

color = 255, 0, 0

elif self.name is not None:

color = 0, 255, 0

cv2.rectangle(frame,

(pos[0] - self.size, pos[1] - self.size),

(pos[0] + self.size, pos[1] + self.size),

color,

1)

cv2.rectangle(frame,

(pos[0] - self.size - 1, pos[1] - self.size - 1),

(pos[0] + self.size + 1, pos[1] + self.size + 1),

(255, 255, 255),

1)

# print(self.getColor(frame, pos))

def move(self, offset):

self.pos = (self.pos[0] + offset[0], self.pos[1] + offset[1])

class Digit:

is\_broken = False

def \_\_init\_\_(self, video):

self.segments = []

self.video = video

self.isNaming = False

self.sorted = {}

self.\_isSorted = False

def sort(self):

self.segments.sort(key=lambda seg: (SN.U, SN.UL, SN.UR, SN.M, SN.BL, SN.BR, SN.B).index(seg.name))

for segment in self.segments:

self.sorted[segment.name] = segment

if segment.name is None:

raise KeyError()

if len(self.sorted) != 7:

raise KeyError

self.\_isSorted = True

def setSegment(self, seg):

self.segments.append(seg)

def scan(self, frame):

data = {}

for seg in self.segments:

data[seg.name] = seg.scan(frame)

self.is\_broken = False

return self.interpret(data), data

@staticmethod

def interpret(data):

return Interrupt.find(data)

def removeLast(self):

self.segments.pop(-1)

def isFull(self):

return len(self.segments) >= 7

def isNamed(self):

return all([s.name is not None for s in self.segments]) and len(self.segments) == 7

class SN(Enum): # Segment Name

U = auto()

UL = auto()

UR = auto()

M = auto()

BL = auto()

BR = auto()

B = auto()

@staticmethod

def getName(i):

return (SN.U, SN.UL, SN.UR, SN.M, SN.BL, SN.BR, SN.B)[i % 7]

class Interrupt:

dataSet = ({SN.U: True, SN.UL: True, SN.UR: True, SN.M: False, SN.BL: True, SN.BR: True, SN.B: True}, # 0

{SN.U: False, SN.UL: False, SN.UR: True, SN.M: False, SN.BL: False, SN.BR: True, SN.B: False}, # 1

{SN.U: True, SN.UL: False, SN.UR: True, SN.M: True, SN.BL: True, SN.BR: False, SN.B: True}, # 2

{SN.U: True, SN.UL: False, SN.UR: True, SN.M: True, SN.BL: False, SN.BR: True, SN.B: True}, # 3

{SN.U: False, SN.UL: True, SN.UR: True, SN.M: True, SN.BL: False, SN.BR: True, SN.B: False}, # 4

{SN.U: True, SN.UL: True, SN.UR: False, SN.M: True, SN.BL: False, SN.BR: True, SN.B: True}, # 5

{SN.U: True, SN.UL: True, SN.UR: False, SN.M: True, SN.BL: True, SN.BR: True, SN.B: True}, # 6

{SN.U: True, SN.UL: False, SN.UR: True, SN.M: False, SN.BL: False, SN.BR: True, SN.B: False}, # 7

{SN.U: True, SN.UL: True, SN.UR: True, SN.M: True, SN.BL: True, SN.BR: True, SN.B: True}, # 8

{SN.U: True, SN.UL: True, SN.UR: True, SN.M: True, SN.BL: False, SN.BR: True, SN.B: True}) # 9

@staticmethod

def find(data):

if data in Interrupt.dataSet:

return True, Interrupt.dataSet.index(data)

else:

errors = []

for number\_data in Interrupt.dataSet:

error = 0

for seg in number\_data:

if data[seg] != number\_data[seg]:

error += 1

errors.append(error)

return False, np.argmin(errors)