**ООО «Цифровые решения»**

УТВЕРЖДАЮ

Генеральный директор

ООО “Цифровые решения”

\_\_\_\_\_\_\_\_\_\_\_\_\_\_Д.В. Матвеев

# *М.П.*

"\_\_\_\_"\_\_\_\_\_\_\_\_\_\_\_\_\_ 2020 г.

ПРОГРАММА ДЛЯ ЭВМ

**SportAISystem 1.0  - программа для спортивных аналитических систем на основе технологий машинного обучения**

Фрагменты исходного текста программы

Листов 16

Авторы:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/Матвеев Д.В./

\_\_\_\_\_\_\_\_\_\_\_\_\_\_/Ивановский Л.И./

© ООО «Цифровые решения», 2020 г.

Ярославль 2020 г.

*combatscounter.py:*

import argparse

import os

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from collections import defaultdict

import operations

plt.rcParams.update({'font.size': 7})

class CombatsCounter():

def \_\_init\_\_(self, markup\_file, out\_dir, human\_number=None):

self.\_\_data = pd.read\_csv(markup\_file, names=['frame', 'id', 'bb\_y', 'bb\_x', 'bb\_h', 'bb\_w'])

self.\_\_ids = list(self.\_\_data['id'].unique())

self.\_\_countframes = max(self.\_\_data['frame'])

self.\_\_countsids = len(self.\_\_ids)

self.\_\_outdirectory = out\_dir

self.\_\_human = human\_number

self.\_\_combatsmatrix = np.zeros((len(self.\_\_ids), len(self.\_\_ids)), dtype=np.int)

def \_\_buildCombatsMatrix(self):

existing\_combats = defaultdict(set) # combats, which were registered on previous frame

for frame\_id in range(1, self.\_\_countframes+1):

# Extract all detected bbox on current frame

frame\_bboxes = self.\_\_data[self.\_\_data['frame'] == frame\_id][['id', 'bb\_y', 'bb\_x', 'bb\_h', 'bb\_w']]

n\_bboxes = len(frame\_bboxes)

for i in range(n\_bboxes-1):

for j in range(i+1, n\_bboxes):

id\_i, id\_j = int(frame\_bboxes.iloc[i]['id']), int(frame\_bboxes.iloc[j]['id'])

if id\_i != id\_j:

if operations.is\_bbox\_intersected(frame\_bboxes.iloc[i], frame\_bboxes.iloc[j]):

if id\_j not in existing\_combats[id\_i]: # combat was not registered

# Add a combat to matrix on symmetric places

idx\_i, idx\_j = self.\_\_ids.index(id\_i), self.\_\_ids.index(id\_j)

self.\_\_combatsmatrix[idx\_i, idx\_j] += 1

self.\_\_combatsmatrix[idx\_j, idx\_i] += 1

# Register new combat

existing\_combats[id\_i].add(id\_j)

existing\_combats[id\_j].add(id\_i)

else:

# stop taking account completed combat

existing\_combats[id\_i].discard(id\_j)

existing\_combats[id\_j].discard(id\_i)

def \_\_buildHumanCombatsDictionary(self):

self.\_\_combatsdict = dict()

for i, comb in enumerate(list(self.\_\_combatsmatrix[self.\_\_ids.index(self.\_\_human)])):

if comb != 0:

self.\_\_combatsdict[self.\_\_ids[i]] = self.\_\_combatsmatrix[i, self.\_\_ids.index(self.\_\_human)]

def \_\_drawCombatsMatrix(self):

fig, ax = plt.subplots(figsize=(12, 9))

plt.subplots\_adjust(left=0.03, bottom=0.03, right=0.97, top=0.97)

im = ax.imshow(self.\_\_combatsmatrix)

ax.set\_xticks(np.arange(self.\_\_countsids))

ax.set\_yticks(np.arange(self.\_\_countsids))

ax.set\_xticklabels(self.\_\_ids)

ax.set\_yticklabels(self.\_\_ids)

ax.tick\_params(top=True, bottom=True, left=True, right=True, labeltop=True, labelright=True)

colorbar = ax.figure.colorbar(im, ax=ax)

colorbar.ax.set\_ylabel('Number of combats', rotation=-90, va='bottom')

for i in range(len(self.\_\_ids)):

for j in range(len(self.\_\_ids)):

ax.text(j, i, self.\_\_combatsmatrix[i, j], ha='center', va='center', color='w')

plt.savefig(os.path.join(self.\_\_outdirectory, 'combats\_matrix.png'))

def \_\_drawBarChartHumanCombats(self):

self.\_\_buildHumanCombatsDictionary()

d = self.\_\_combatsdict

if len(d) != 0:

fig, ax = plt.subplots(figsize=(12, 9))

plt.subplots\_adjust(left=0.04, bottom=0.04, right=0.96, top=0.96)

plt.grid()

ax.bar(np.arange(len(d)), list(d.values()), zorder=2)

ax.set\_xticks(np.arange(len(d)))

ax.set\_xticklabels(list(map(str, d.keys())))

ax.set\_title('Covered distances by players')

ax.set\_ylabel('Pixels')

fig.savefig(os.path.join(self.\_\_outdirectory, 'combats\_\_\_human\_{}.png'.format(self.\_\_human)))

def calculateCombatsStatistics(self):

self.\_\_buildCombatsMatrix()

if self.\_\_human is None:

self.\_\_drawCombatsMatrix()

else:

self.\_\_buildHumanCombatsDictionary()

self.\_\_drawBarChartHumanCombats()

def init\_argparse():

'''

Initializes argparse

'''

parser = argparse.ArgumentParser(description='Background extraction from the video')

parser.add\_argument(

'--markup',

nargs='?',

help='Markup file',

required=True,

type=str)

parser.add\_argument(

'--out\_dir',

nargs='?',

help='Output directory for saving files with calculated statistics',

required=True,

type=str)

parser.add\_argument(

'--human',

nargs='?',

help='Number of sportsman',

default=None,

type=int)

return parser

def main():

parser = init\_argparse()

# Extract arguments of script

args = parser.parse\_args()

# Calculate statistics about combats

comb\_acc = CombatsCounter(args.markup, args.out\_dir, args.human)

comb\_acc.calculateCombatsStatistics()

if \_\_name\_\_ == '\_\_main\_\_':

main()

*emailsending.py:*

import argparse

import os

import yagmail

import constants

class EMailSending:

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

self.\_\_fromuser = constants.SENT\_FROM\_USER

self.\_\_frompassword = constants.SENT\_FROM\_PASSWORD

def sendEMail(self, to\_users: list, attached\_files\_paths: list):

smtp\_connection = yagmail.SMTP(user=self.\_\_fromuser, password=self.\_\_frompassword,

host='smtp.gmail.com') # connect to SMTP server

email\_subject = 'SportAISystem 1.0 Statistics'

smtp\_connection.send(to\_users, email\_subject, attached\_files\_paths) # send the email

def init\_argparse():

'''

Initializes argparse

'''

parser = argparse.ArgumentParser(description='Sending statistics by the use of e-mail')

parser.add\_argument(

'--to',

nargs='?',

help='E-mail of recipient',

required=True,

type=str)

parser.add\_argument(

'--folder',

nargs='?',

help='Folder with files to send',

required=True,

type=str)

return parser

def main():

parser = init\_argparse()

args = parser.parse\_args() # extract arguments of script

sending = EMailSending()

# Collect sending files

content = [os.path.join(os.path.abspath(args.folder), file) for file in os.listdir(args.folder)]

sending.sendEMail([args.to], content)

if \_\_name\_\_ =='\_\_main\_\_':

main()

*motionheatmap.py:*

import argparse

import os

import pandas as pd

from PIL import Image

import constants

import operations

from heatmapper import Heatmapper

from traceplace import Traceplace

class MotionHeatmap():

def \_\_init\_\_(self, markup\_file, out\_dir, human\_number=None, marker\_pos='lower\_center'):

self.\_\_data = pd.read\_csv(markup\_file, names=['frame', 'id', 'bb\_y', 'bb\_x', 'bb\_h', 'bb\_w'])

self.\_\_outdirectory = out\_dir

self.\_\_background = Image.open(constants.BACKGROUND\_READY\_IMAGE)

self.\_\_human = human\_number

self.\_\_markerpos = Traceplace[str(marker\_pos).upper()]

def \_\_loadPoints(self):

if self.\_\_human is not None:

self.\_\_points = [operations.get\_point(row, self.\_\_markerpos)

for \_, row in self.\_\_data[self.\_\_data['id'] == self.\_\_human].iterrows()]

else:

self.\_\_points = [operations.get\_point(row, self.\_\_markerpos) for \_, row in self.\_\_data.iterrows()]

def buildHeatmap(self):

self.\_\_loadPoints()

heatmapper = Heatmapper()

heatmap\_img = heatmapper.buildHeatmapOnImage(self.\_\_points, self.\_\_background)

if self.\_\_human is not None:

heatmap\_img.save(os.path.join(self.\_\_outdirectory, 'heatmap\_\_\_human\_{}.png'.format(self.\_\_human)))

else:

heatmap\_img.save(os.path.join(self.\_\_outdirectory, 'heatmap.png'))

def init\_argparse():

'''

Initializes argparse

'''

parser = argparse.ArgumentParser(description='Background extraction from the video')

parser.add\_argument(

'--markup',

nargs='?',

help='Markup file',

required=True,

type=str)

parser.add\_argument(

'--outfile',

nargs='?',

help='Output file to save heatmap',

required=True,

type=str)

parser.add\_argument(

'--human',

nargs='?',

help='Number of sportsman',

default=None,

type=int)

parser.add\_argument(

'--traceplace',

nargs='?',

help='Place of marker on the bbox where the trace is drawing',

default='lower\_center',

type=str)

return parser

def main():

parser = init\_argparse()

# Extract arguments of script

args = parser.parse\_args()

# Building heatmap of motions

mh = MotionHeatmap(markup\_file=args.markup, out\_file=args.out\_dir,

human\_number=args.human, marker\_pos=args.traceplace)

mh.buildHeatmap()

if \_\_name\_\_ == '\_\_main\_\_':

main()

*motiontrajectories.py*

import argparse

import cv2

import json

import os

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from PIL import Image

from collections import defaultdict

from scipy.spatial import distance

import constants

import operations

from traceplace import Traceplace

TRAJECTORY\_IMAGE = 'trajectories.png'

class MotionTrajectories():

def \_\_init\_\_(self, markup\_file, out\_dir, human\_number=None, marker\_pos='lower\_center'):

self.\_\_data = pd.read\_csv(markup\_file, names=['frame', 'id', 'bb\_y', 'bb\_x', 'bb\_h', 'bb\_w'])

self.\_\_human = human\_number

if self.\_\_human is not None:

self.\_\_data = self.\_\_data[self.\_\_data['id'] == self.\_\_human]

self.\_\_outdirectory = out\_dir

self.\_\_background = np.asarray(Image.open(constants.BACKGROUND\_READY\_IMAGE))

self.\_\_markerpos = Traceplace[str(marker\_pos).upper()]

self.\_\_distances = defaultdict(int)

def calculateTraceStatistics(self):

self.\_\_drawTrajectories()

self.\_\_saveResults()

def \_\_drawTrajectories(self):

prev\_point = dict()

for \_, row in self.\_\_data.iterrows():

human\_id = int(row['id'])

color = operations.get\_color(human\_id)

point = operations.get\_point(row, self.\_\_markerpos)

if int(human\_id) not in prev\_point.keys():

self.\_\_background = cv2.circle(self.\_\_background, point, radius=3, color=color, thickness=5)

prev\_point[human\_id] = point

else:

self.\_\_background = cv2.line(self.\_\_background, point, prev\_point[human\_id], color=color, thickness=2)

self.\_\_distances[human\_id] += distance.euclidean(point, prev\_point[human\_id])

prev\_point[human\_id] = point

def \_\_drawBarChartDistances(self):

fig, ax = plt.subplots(figsize=(12, 9))

plt.rcParams.update({'font.size': 8})

plt.subplots\_adjust(left=0.03, bottom=0.03, right=0.97, top=0.97)

plt.grid()

d = self.\_\_distances

ax.barh(np.arange(len(d)), list(d.values()), zorder=2)

ax.set\_yticks(np.arange(len(d)))

ax.set\_yticklabels(list(map(str, d.keys())))

ax.invert\_yaxis() # labels read top-to-bottom

ax.set\_title('Covered distances by players')

ax.set\_xlabel('Pixels')

fig.savefig(os.path.join(self.\_\_outdirectory, 'covered\_distances.png'))

def \_\_saveResults(self):

Image.fromarray(self.\_\_background).save(os.path.join(self.\_\_outdirectory, TRAJECTORY\_IMAGE))

if self.\_\_human is not None:

json\_filename = 'covered\_distance\_\_\_human\_{}.json'.format(self.\_\_human)

with open(os.path.join(self.\_\_outdirectory, json\_filename), 'w') as fp:

json.dump(self.\_\_distances, fp)

else:

self.\_\_drawBarChartDistances()

def init\_argparse():

'''

Initializes argparse

'''

parser = argparse.ArgumentParser(description='Background extraction from the video')

parser.add\_argument(

'--markup',

nargs='?',

help='Markup file',

required=True,

type=str)

parser.add\_argument(

'--out\_dir',

nargs='?',

help='Output directory for saving files with calculated statistics',

required=True,

type=str)

parser.add\_argument(

'--human',

nargs='?',

help='Number of sportsman',

type=int)

parser.add\_argument(

'--traceplace',

nargs='?',

help='Place of marker on the bbox where the trace is drawing',

default='lower\_center',

type=str)

return parser

def main():

parser = init\_argparse()

# Extract arguments of script

args = parser.parse\_args()

# Calculate statistics about trajectories

mt = MotionTrajectories(markup\_file=args.markup, out\_dir=args.out\_dir,

human\_number=args.human, marker\_pos=args.traceplace)

mt.calculateTraceStatistics()

if \_\_name\_\_ == '\_\_main\_\_':

main()

*operations.py:*

from traceplace import Traceplace

def time\_string(ms):

seconds = int((ms/1000) % 60)

minutes = int((ms/(1000\*60)) % 60)

return '{:d}:{:02d}'.format(minutes, seconds)

def get\_color(human\_number):

color = ((111\*human\_number)%255, (51\*human\_number)%255, (87\*human\_number)%255)

return color

def get\_point(row\_dataframe, marker\_pos):

if marker\_pos == Traceplace.LOWER\_LEFT:

return (int(row\_dataframe['bb\_y']),

int(row\_dataframe['bb\_x'] + row\_dataframe['bb\_w']))

if marker\_pos == Traceplace.LOWER\_CENTER:

return (int(row\_dataframe['bb\_y'] + row\_dataframe['bb\_h'] / 2.0),

int(row\_dataframe['bb\_x'] + row\_dataframe['bb\_w']))

if marker\_pos == Traceplace.LOWER\_RIGHT:

return (int(row\_dataframe['bb\_y'] + row\_dataframe['bb\_h']),

int(row\_dataframe['bb\_x'] + row\_dataframe['bb\_w']))

if marker\_pos == Traceplace.UPPER\_LEFT:

return (int(row\_dataframe['bb\_y']),

int(row\_dataframe['bb\_x']))

if marker\_pos == Traceplace.UPPER\_CENTER:

return (int(row\_dataframe['bb\_y'] + row\_dataframe['bb\_h'] / 2.0),

int(row\_dataframe['bb\_x']))

if marker\_pos == Traceplace.UPPER\_RIGHT:

return (int(row\_dataframe['bb\_y'] + row\_dataframe['bb\_h']),

int(row\_dataframe['bb\_x']))

if marker\_pos == Traceplace.CENTER:

return (int((row\_dataframe['bb\_y'] + row\_dataframe['bb\_h']) / 2.0),

int((row\_dataframe['bb\_x'] + row\_dataframe['bb\_w']) / 2.0))

def is\_bbox\_intersected(cur\_bbox, other\_bbox):

bbox\_y1 = cur\_bbox['bb\_y']

bbox\_y2 = cur\_bbox['bb\_y'] + cur\_bbox['bb\_h']

bbox\_x1 = cur\_bbox['bb\_x']

bbox\_x2 = cur\_bbox['bb\_x'] + cur\_bbox['bb\_w']

y1 = other\_bbox['bb\_y']

y2 = other\_bbox['bb\_y'] + other\_bbox['bb\_h']

x1 = other\_bbox['bb\_x']

x2 = other\_bbox['bb\_x'] + other\_bbox['bb\_w']

if (x1 > bbox\_x1 and x1 < bbox\_x2 and y1 > bbox\_y1 and y1 < bbox\_y2) or \

(x2 > bbox\_x1 and x2 < bbox\_x2 and y2 > bbox\_y1 and y2 < bbox\_y2) or \

(x1 > bbox\_x1 and x1 < bbox\_x2 and y2 > bbox\_y1 and y2 < bbox\_y2) or \

(x2 > bbox\_x1 and x2 < bbox\_x2 and y1 > bbox\_y1 and y1 < bbox\_y2):

return True

return False

*statdialog.py:*

import os

import shutil

from PyQt5.QtWidgets import QDialog, QCheckBox, QVBoxLayout, QHBoxLayout, QRadioButton, QGroupBox, QSpinBox, \

QLineEdit, QLabel, QPushButton, QSplitter, QFileDialog, QSizePolicy

from PyQt5.QtGui import QIcon

from PyQt5.QtCore import Qt

from PyQt5.QtWidgets import QApplication

import constants

from combatscounter import CombatsCounter

from emailsending import EMailSending

from motionheatmap import MotionHeatmap

from motiontrajectories import MotionTrajectories

class StatDialog(QDialog):

def \_\_init\_\_(self, parent=None, markup=None, width=1024, height=768):

super(StatDialog, self).\_\_init\_\_(parent)

self.\_\_markup = markup

self.\_\_dirname = ''

self.setWindowIcon(QIcon('Icons/statistics.png'))

self.calculatePushButton = QPushButton('Calculate')

self.calculatePushButton.clicked.connect(self.calculateStatistics)

self.errorLabel = QLabel('')

self.errorLabel.setSizePolicy(QSizePolicy.Preferred, QSizePolicy.Maximum)

self.heatmapCheckBox = QCheckBox('Heatmap', self)

self.heatmapCheckBox.setToolTip('Motion heatmap on the field')

self.heatmapCheckBox.setChecked(True)

self.combatsCheckBox = QCheckBox('Combats', self)

self.combatsCheckBox.setToolTip('Number of combats between rivals')

self.pathsCheckBox = QCheckBox('Motion paths', self)

self.pathsCheckBox.setToolTip('Motion trajectories and their lenght')

self.allRadioButton = QRadioButton('For all people')

self.allRadioButton.setChecked(True)

self.personRadioButton = QRadioButton('For specific person')

self.humanSpinBox = QSpinBox()

self.humanSpinBox.setMinimum(1)

self.allRadioButton.toggled.connect(self.allRadioButtonToggled)

self.personRadioButton.toggled.connect(self.personRadioButtonToggled)

self.allRadioButtonToggled()

self.localRadioButton = QRadioButton('Local saving')

self.localRadioButton.setChecked(True)

self.savepathLabel = QLabel('Please choose directory')

self.savepathLabel.setToolTip('')

self.savepathButton = QPushButton('...')

self.savepathButton.clicked.connect(self.chooseDirectory)

self.savepathButton.setFixedWidth(25)

saveLayout = QHBoxLayout()

saveLayout.addWidget(self.savepathLabel)

saveLayout.addWidget(self.savepathButton)

self.remoteRadioButton = QRadioButton('E-mail')

self.emailLineEdit = QLineEdit()

statGroupBox = QGroupBox('Sport statistics')

statLayout = QVBoxLayout()

statLayout.addWidget(self.heatmapCheckBox)

statLayout.addWidget(self.combatsCheckBox)

statLayout.addWidget(self.pathsCheckBox)

statLayout.addStretch(1)

statGroupBox.setLayout(statLayout)

optionsGroupBox = QGroupBox('Options')

optionsLayout = QVBoxLayout()

optionsLayout.addWidget(self.allRadioButton)

optionsLayout.addWidget(self.personRadioButton)

optionsLayout.addWidget(self.humanSpinBox)

optionsLayout.addStretch(1)

optionsGroupBox.setLayout(optionsLayout)

sendGroupBox = QGroupBox('Sending', alignment=Qt.AlignHCenter)

sendLayout = QVBoxLayout()

sendLayout.addWidget(self.localRadioButton)

sendLayout.addLayout(saveLayout)

sendLayout.addWidget(self.remoteRadioButton)

sendLayout.addWidget(self.emailLineEdit)

sendGroupBox.setLayout(sendLayout)

vertLayout = QHBoxLayout()

vertLayout.addWidget(statGroupBox)

vertLayout.addWidget(optionsGroupBox)

mainLayout = QVBoxLayout()

mainLayout.addLayout(vertLayout)

mainLayout.addWidget(QSplitter(Qt.Horizontal))

mainLayout.addWidget(sendGroupBox)

mainLayout.addWidget(QSplitter(Qt.Horizontal))

mainLayout.addWidget(self.calculatePushButton)

mainLayout.addWidget(self.errorLabel)

self.setLayout(mainLayout)

self.setMinimumWidth(350)

self.setWindowTitle('Videoanalysis')

if not os.path.exists(constants.STATISTICS\_FOLDER):

os.makedirs(constants.STATISTICS\_FOLDER)

self.setLocalDirectory(constants.STATISTICS\_FOLDER)

def allRadioButtonToggled(self):

self.humanSpinBox.setDisabled(True)

self.humanSpinBox.setToolTip('')

def personRadioButtonToggled(self):

self.humanSpinBox.setDisabled(False)

self.humanSpinBox.setToolTip('Choose the number of human')

def setLocalDirectory(self, str\_path):

self.\_\_dirname = str\_path

if len(self.\_\_dirname) > 44:

reduced\_dirname = '...{}'.format(str(self.\_\_dirname)[-41:])

else:

reduced\_dirname = str(self.\_\_dirname)

self.savepathLabel.setText(reduced\_dirname)

self.savepathLabel.setToolTip(str(self.\_\_dirname))

def chooseDirectory(self):

self.\_\_dirname = QFileDialog.getExistingDirectory(self, 'Choose directory', os.getcwd(), QFileDialog.ShowDirsOnly)

if self.\_\_dirname:

self.setLocalDirectory(self.\_\_dirname)

self.errorLabel.setText('')

def calculateStatistics(self):

if not (self.heatmapCheckBox.isChecked() or self.pathsCheckBox.isChecked() or self.combatsCheckBox.isChecked()):

self.errorLabel.setText('Please choose almost one value of statistics to calculate')

else:

if self.\_\_markup:

if self.\_\_dirname:

statdir = os.path.join(self.\_\_dirname, os.path.splitext(os.path.basename(self.\_\_markup))[0])

if not os.path.exists(statdir):

os.makedirs(statdir)

else:

shutil.rmtree(statdir)

if self.personRadioButton.isChecked():

human = self.humanSpinBox.value()

else:

human = None

QApplication.setOverrideCursor(Qt.WaitCursor)

if self.heatmapCheckBox.isChecked():

mh = MotionHeatmap(markup\_file=self.\_\_markup, out\_dir=statdir, human\_number=human)

mh.buildHeatmap()

if self.pathsCheckBox.isChecked():

mt = MotionTrajectories(markup\_file=self.\_\_markup, out\_dir=statdir, human\_number=human)

mt.calculateTraceStatistics()

if self.combatsCheckBox.isChecked():

comb\_acc = CombatsCounter(markup\_file=self.\_\_markup, out\_dir=statdir, human\_number=human)

comb\_acc.calculateCombatsStatistics()

if self.remoteRadioButton.isChecked():

email = self.emailLineEdit.text()

if email:

sending = EMailSending()

content = [os.path.join(os.path.abspath(statdir), file) for file in os.listdir(statdir)]

sending.sendEMail([email], content)

self.errorLabel.setText('Files with statistics values were successfully sent by e-mail')

else:

self.errorLabel.setText('Please write down email address')

else:

self.errorLabel.setText('Files with statistics values were loaded in local directory')

QApplication.restoreOverrideCursor()

else:

self.errorLabel.setText('Please choose the directory to save')

else:

self.errorLabel.setText('Cannot load the markup file')

*traceplace.py:*

from enum import Enum

class Traceplace(Enum):

LOWER\_LEFT = 0,

LOWER\_CENTER = 1,

LOWER\_RIGHT = 2,

UPPER\_LEFT = 3,

UPPER\_CENTER = 4,

UPPER\_RIGHT = 5,

CENTER = 6

def \_\_repr\_\_(self):

if self.value == Traceplace.LOWER\_LEFT:

return 'lower\_left'

if self.value == Traceplace.LOWER\_CENTER:

return 'lower\_center'

if self.value == Traceplace.LOWER\_RIGHT:

return 'lower\_right'

if self.value == Traceplace.UPPER\_LEFT:

return 'upper\_left'

if self.value == Traceplace.UPPER\_CENTER:

return 'upper\_center'

if self.value == Traceplace.UPPER\_RIGHT:

return 'upper\_right'

if self.value == Traceplace.CENTER:

return 'center'

*heatmapper.py:*

import numpy as np

from matplotlib.colors import LinearSegmentedColormap

from PIL import Image

import constants

class Heatmapper:

def \_\_init\_\_(self, point\_diameter=30, point\_strength=0.2, opacity=0.5):

self.point\_diameter = point\_diameter

self.point\_strength = point\_strength

self.opacity = opacity

self.cmap = self.\_\_setColorMapFromImage(constants.COLORMAP\_IMAGE)

def \_\_imageToOpacity(self, img, opacity):

img = img.copy()

alpha = img.split()[3]

img.putalpha(alpha.point(lambda p: int(p \* opacity)))

return img

def buildHeatmapOnImage(self, points, background\_img):

width, height = background\_img.size

heatmap = self.\_\_makeHeatmap(width, height, points)

heatmap = self.\_\_imageToOpacity(heatmap, self.opacity)

if background\_img is not None:

return Image.alpha\_composite(background\_img.convert('RGBA'), heatmap)

else:

return heatmap

def \_\_setColorMapFromImage(self, colormap\_img):

# load colomap image

img = Image.open(colormap\_img)

img = img.resize((256, img.height))

# extract colors from colormap image

colours = [img.getpixel((x, 0)) for x in range(256)]

colours = [(r/255, g/255, b/255, a/255) for r, g, b, a in colours]

return LinearSegmentedColormap.from\_list('from\_image', colours)

def \_\_makeHeatmap(self, width, height, points):

heatmap = Image.new('L', (width, height), color=255) # empty heatmap

spot = Image.open(constants.SPOT\_IMAGE).copy().resize((self.point\_diameter, self.point\_diameter),

resample=Image.ANTIALIAS)

spot = self.\_\_imageToOpacity(spot, self.point\_strength)

for x, y in points: # locate spots on the heatmap

x, y = int(x - self.point\_diameter/2), int(y - self.point\_diameter/2)

heatmap.paste(spot, (x, y), spot)

res = self.cmap(np.array(heatmap), bytes=True) # paint over using heatmap

return Image.fromarray(res)

*main.py*

from PyQt5.QtWidgets import QApplication

import sys

from PyQt5.QtGui import QIcon

from videoplayer import VideoPlayer

if \_\_name\_\_ == '\_\_main\_\_':

app = QApplication(sys.argv)

app.setWindowIcon(QIcon('Icons/voleyball-player.png'))

player = VideoPlayer(width=800, height=600)

player.show()

sys.exit(app.exec\_())