Augmented Maps

Luis Coelho
up201304273@fe.up.pt
Manuel Gomes
up201402679@fe.up.pt

Faculdade de Engenharia
University of Porto
Porto, Portugal

1 Introduction

The most common way to perform Augmented Reality (AR) is to use markers to calibrate the camera pose, which enables later projection of virtual content into the captured real scene accordingly. However, marker-based AR has an inherent drawback, i.e. the requirement of specific marker(s), which may limit the development of its applications. In contrast, markerless AR uses visual or depth information of the captured scene to estimate the camera pose, thus no special marker is needed for calibration, nonetheless, requires higher computation complexity. In this work, we design 2 programs, Preparation and Calibration, one able to prepare an image map, where its points if interested are associated, and another that is able to augment either a static image or video frame, after making a match with an image map prepared.

2 Proposed Solution

The proposed solution is divided in five 2 major steps, Preparation of the image map, where the user marks the desired points of interest, and the Augmentation of an image, either through video or a static image, where the in case the there is an image map in database matching the current frame, the nearest point of interest is rendered. The solution was developed using Python and OpenCV and NumPy as main librarys, while the GUI was developed with PyQT5. In each subsection is described in detail every step and decisions made, supported by the reasoning behind them. The source code is available in the appendix.

3 Preparation

In this step, the user starts by selecting an image of a map and marks one or more points of interest. At each new point of interest, a rectangle is created and the user must write the name of it and also add one or more images associated with the point of interest. The user also has the option to move the rectangle, as well as delete the point of interest created. Before saving the prepared map, the user must also give a name to it, as well as give the scale of the map which will be used in the Augmentation step to calculate the distance between the center and the nearest Point of Interest. When the prepared map is saved, a key-points and feature calculation is made using the SIFT algorithm. Histogram equalization method is adopted to preprocess the original image, using CLAHE, to enhance the useful information. Then the preprocessed image is used to the SIFT algorithm to achieve the extraction and matching of the image feature points. The purpose of image preprocessing is to increase the matching number of image feature points, and improve the matching rate of image feature

4 Augmentation

In this phase the use either selects a static image or turns on the video camera and augments the map that is passed. The main loop of the program is get a video frame of the camera (or a static image), estimate the position and orientation of the camera, detect and recognize features, render the augmented scene in case a match was found. At each video frame, the features and key-points from it are calculated using a SIFT algorithm after the preprocessing with histogram equalization to enhance contrast. After that, it is attempted to match the the frame with the prepared images in the database. For that is used FLANN which stands for Fast Library for Approximate Nearest Neighbors. It contains a collection of algorithms optimized for fast nearest neighbor search in large datasets and for high dimensional features. It works faster than BFMatcher for large datasets. For FLANN based matcher, it is need to pass two dictionaries which specifies the algorithm to be used, its related parameters etc. First one is IndexParams. The second dictionary is the SearchParams. It specifies the number of times the trees in the index should be recursively traversed. Higher values gives better precision, but also takes more time. The value for searchParams used was 50, while for indexParams was used the value 5 for the parameter TREES. After that, a ratio test as per Lowe's paper is used in order to detect the good matches. In case there are at least 80 good matches, the prepared image tested is considered a good image. After all the images are compared, they are sorted by the number of good matches and the one with most number of good matches is selected. The frame is then augmented using the selected image. The first thing done is calculate the homography between the frame and the prepared image. The homography is calculated using RANSAC with a value of 5. Using the matrix calculated, the end position of each point of interest point is calculated. With the end corners calculated, using the Euclidian distance, the nearest point of interest of the center is calculated. It is then drawn the center of the map, a compass pointing to the North, using the matrix calculated in the homography step. Also, using the scale of the map, the real distance from the nearest point of interest to the center is calculated and then, the name, distance and an image associated to the nearest point of interest are drawn in one of the corners of the map. In the end, using the camera parameters from the calibration of the camera, a projection matrix is calculated, with which is rendered a 3D pyramid in the frame, representing the nearest point of interest.

5 Conclusions and Further Improvements

Key-points and features and detection is one of the most critical steps in of the whole process. The use o SIFT algorithm proved to have a quite good accuracy, although is not the best in terms of performance, as is possible to see in the augmentation phase through camera video. Having this in regard, we think in future work would be important to try a different approach in this phase or even using a different algorithm just in this phase.

A main.py

```
import sys
from PyQt5 import QtWidgets
from augmented_maps import AugmentedMaps


def main():
    print ('Argument␣List:␣' + str(sys.argv))
    app = QtWidgets.QApplication(sys.argv)
    if len(sys.argv) > 1:
        window = AugmentedMaps(True)
    else:
        window = AugmentedMaps(False)
    sys.exit(app.exec_())

if __name__ == '__main__':
    main()
```

B utils.py

```
import cv2
import numpy as np
import math
import yaml

from typing import Tuple, List
from PyQt5 import QtGui as gui
```

```
# FLANN parameters
FLANN_INDEX_KDTREE = 1
index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
search_params = dict(checks=50)


# Draws center circle in the map
def draw_center_map(img, width, height):
    cv2.circle(img, (int(width/2), int(height/2)),
               6, (0), 2, lineType=cv2.LINE_AA)
    cv2.circle(img, (int(width/2), int(height/2)),
               6, (255, 255, 0), −1, lineType=cv2.LINE_AA)

    return img


# Return nearest Point of Interest
def get_nearest_interestpoint(image_prepared, matrix, w, h):
    nearest_interestpoint = [None, None, None, None]

    for interestpoint in image_prepared.interestPoints:

        print(
            f"Points␣of␣Interest␣−␣Coord␣Xi:␣{interestpoint.x}")
        print(
            f"Points␣of␣Interest␣−␣Coord␣Yi:␣{interestpoint.y}")
        print(
            f"Points␣of␣Interest␣−␣Width:␣{interestpoint.w}")
        print(
            f"Points␣of␣Interest␣−␣Height:␣{interestpoint.h}")

        # Gets the corners of the interestpoint
        pts_interestpoint = np.float32([[interestpoint.x, interestpoint.y],
             [interestpoint.x, interestpoint.y + interestpoint.h − 1], [
             interestpoint.x +
```

```
        # Project corners into frame
        dst_interestpoint = cv2.perspectiveTransform(
            pts_interestpoint, matrix)

        # Calculares centroid of the transformed interestpoint
        centroid_interestpoint = get_centroid(
            (dst_interestpoint[0][0], dst_interestpoint[1][0],
                 dst_interestpoint[2][0], dst_interestpoint[3][0]))

        # Calculates distance between the
        distance = distante_between_points(
            centroid_interestpoint, (w/2, h/2))

        print(distance)

        if nearest_interestpoint[1] is None or distance <
             nearest_interestpoint[1]:
            print("Changed␣nearest␣Point␣of␣Interest")
            nearest_interestpoint = [
                dst_interestpoint, distance, interestpoint.name,
                     interestpoint.images[0]]
                                                              intere
    return nearest_interestpoint


# Returns the compass points
def get_compass_points(width, height):
    pts = np.float32(
        [[int(width/2) + 30, int(height/2)], [int(width/2) + 40, int(
             height/2) + 30], [int(width/2) + 50, int(height/2)], [int(
             width/2) + 40, int(height/2) − 30]]).reshape(−1, 1, 2)

    return pts


# Returns the header image points
def get_header_points(xi, yi, w):
    pts = np.float32(
        [[xi, yi − 30], [xi, yi], [w, yi], [w, yi − 30]]).reshape(−1, 1, 2)

    return pts


# Get descriptors from an image
def get_features(img) −> Tuple[List[cv2.KeyPoint], np.ndarray]:
    sift = cv2.xfeatures2d.SIFT_create()
    return sift.detectAndCompute(img, None)


# Match descriptors between 2 images
def match_descriptors(src_des: np.ndarray, target_des: np.ndarray):
    flann = cv2.FlannBasedMatcher(index_params, search_params)
    matches = flann.knnMatch(src_des, target_des, k=2)

    # Ratio test as per Lowe's paper
    good = []
    for m, n in matches:
        if m.distance < 0.7 ∗ n.distance:
            good.append(m)
    return good


# Histogram equalization
def histogram_equalization(img):
    clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(2, 2))
    img_grayscale = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    img_hist_eq = clahe.apply(img_grayscale)
    return img_hist_eq


# Converts keypoints to a dictionary to allow its serialization
def keypoints_to_kpdict(kps):
    keypoints = []
    for keypoint in kps:
        temp: dict = {
            'pt': keypoint.pt,
            'angle': keypoint.angle,
            'response': keypoint.response,
            'octave': keypoint.octave,
            'class_id': keypoint.class_id
        }

        keypoints.append(temp)

    return keypoints


# Calculates centroid of the polygon
def get_centroid(vertexes):
    print("Calculating␣the␣centroid␣of␣a␣polygon␣representing␣a␣
         point␣of␣interest")
    _x_list = [vertex[0] for vertex in vertexes]
    _y_list = [vertex[1] for vertex in vertexes]
    _len = len(vertexes)
    _x = sum(_x_list) / _len
    _y = sum(_y_list) / _len
    return(_x, _y)


# Calculates distance between 2 points
def distante_between_points(p1, p2):
    print("Calculating␣the␣distance␣between␣a␣point␣of␣interest␣
         and␣the␣center")
    distance = math.sqrt(((p1[0]−p2[0])∗∗2)+((p1[1]−p2[1])∗∗2))
    return distance


# Converts qimage to n umpy
def qimage_to_numpy(image: gui.QImage):
    ptr = image.bits()
    w, h, _ = image.width(), image.height(), image.depth()
    ptr.setsize(w ∗ h ∗ 4)
    return np.array(ptr).reshape(h, w, 4)


# Converts numpy to qimage
def numpy_to_qimage(src: np.array):
    shape = src.shape
    h, w = shape[0], shape[1]
    d = 1
    if len(shape) == 3:
        d = shape[2]
    return gui.QImage(src, w, h, w ∗ d, gui.QImage.Format_RGB888
         if d != 1 else gui.QImage.Format_Grayscale8)


# Converts an image to qimage
def image_to_qimage(img):
    img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    return numpy_to_qimage(img_rgb)


def camera_calibration_matrix():
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.
         TERM_CRITERIA_MAX_ITER, 30, 0.001)

    objp = np.zeros((6∗7,3), np.float32)
    objp[:,:2] = np.mgrid[0:7,0:6].T.reshape(−1,2)

    objpoints = []
    imgpoints = []

    cap = cv2.VideoCapture(0)
    found = 0
    while(found < 30):
        ret, img = cap.read()
        img = cv2.flip(img, 1)
        cv2.imwrite('photo.png' , img)
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        ret, corners = cv2.findChessboardCorners(gray, (7,6), None)

        if ret == True:
            objpoints.append(objp)
            corners2 = cv2.cornerSubPix(gray, corners, (11,11),
                 (−1,−1), criteria)
            imgpoints.append(corners2)

            img = cv2.drawChessboardCorners(img, (7,6), corners2,
                 ret)
            found += 1

        cv2.imshow('img', img)
        cv2.waitKey(10)
        if found == 30:
            cv2.imwrite ('output.png', img)
    cap.release()
    cv2.destroyAllWindows()

    ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints,
         imgpoints, gray.shape [::−1], None, None)

    data = {'camera_matrix': np.asanyarray(mtx), 'dist_coeff': np.
         asarray(dist)}

    with open("camera_parameters.yaml", "w") as f:
        yaml.dump(data, f)

def projection_matrix(camera_parameters, homography):

    # Compute rotation along the x and y axis as well as the translation
    homography = homography ∗ (−1)
    rot_and_transl = np.dot(np.linalg.inv(camera_parameters),
         homography)
    col_1 = rot_and_transl[:, 0]
    col_2 = rot_and_transl[:, 1]
    col_3 = rot_and_transl[:, 2]
    # normalise vectors
    l = math.sqrt(np.linalg.norm(col_1, 2) ∗ np.linalg.norm(col_2, 2))
    rot_1 = col_1 / l
    rot_2 = col_2 / l
    translation = col_3 / l
    # compute the orthonormal basis
    c = rot_1 + rot_2
    p = np.cross(rot_1, rot_2)
    d = np.cross(c, p)
    rot_1 = np.dot(c / np.linalg.norm(c, 2) + d / np.linalg.norm(d, 2), 1 /
         math.sqrt(2))
    rot_2 = np.dot(c / np.linalg.norm(c, 2) − d / np.linalg.norm(d, 2), 1 /
            math.sqrt(2))
    rot_3 = np.cross(rot_1, rot_2)
    # finally, compute the 3D projection matrix from the model to the
          current frame
    projection = np.stack((rot_1, rot_2, rot_3, translation)).T
    return np.dot(camera_parameters, projection)

def render(img, projection, w = 100, h = 100):

    scale_matrix = np.eye(3) ∗ 12

    faces = np.float32([ [[−3,−3,0], [0,0,20], [3,−3,0]], [[−3,3,0],
          [0,0,20], [−3,−3,0]], [[3,3,0], [0,0,20], [−3,3,0]], [[3,−3,0],
          [0,0,20], [3,3,0]]])


    for face in faces:
        points = np.dot(face, scale_matrix)
        points = np.array([[p[0] + w / 2, p[1] + h / 2, p[2]] for p in
              points])
        dst = cv2.perspectiveTransform(points.reshape(−1, 1, 3),
              projection)
        imgpts = np.int32(dst)
        cv2.fillConvexPoly(img, imgpts, (137, 27, 211))





    return img
```

C augmented$_m$*aps.py*

```
import os

import cv2
import numpy as np
import keyboard
import yaml
import _thread
from PyQt5 import (QtWidgets as qt,
                   QtGui as gui,
                   QtCore as qtc)
from PyQt5.QtCore import Qt

import utils
from interest_point_augment_graphic import
    InterestPointAugmentGraphic
from image_map import ImageMap
from database import Database
from preparation import Preparation


class AugmentedMaps(qt.QMainWindow):

    MTX = None
    DIST = None
    debug = False

    def __init__(self, debug):
        super().__init__()
        self.database: Database = None
        self.database = Database.connect(’db.db’)
        self.configure_window()
        self.configure_menu()
        self.__entryWindow = None
        self.scene = qt.QGraphicsScene()
        self.view = qt.QGraphicsView(self.scene)
        self.popup_list: EntriesList = None
        self.setCentralWidget(self.view)
        self.entry = None
        AugmentedMaps.debug = debug
        self.show()

    def configure_menu(self):
        menubar = self.menuBar()
        menubar.setNativeMenuBar(False)

        file_menu = menubar.addMenu(’Augmentation’)

        open_act = qt.QAction(’Augment␣Map’, self)
        open_act.triggered.connect(self.open_image_map)
        file_menu.addAction(open_act)

        capture_video = qt.QAction(’Capture␣Video’, self)
        capture_video.triggered.connect(self.open_capture)
        file_menu.addAction(capture_video)

        calibrate_camera = qt.QAction(’Calibrate␣Camera’, self)
        calibrate_camera.triggered.connect(self.
              open_camera_calibration)
        file_menu.addAction(calibrate_camera)

        exit_action = qt.QAction(’Quit’, self)
        exit_action.triggered.connect(qt.qApp.quit)
        file_menu.addAction(exit_action)

        menubar.addAction(file_menu.menuAction())

        database_menu = menubar.addMenu(’Preparation’)

        add_database_action = qt.QAction(’Add␣Map’, self)
        add_database_action.triggered.connect(self.
              open_add_entry_window)
        database_menu.addAction(add_database_action)

        list_entries_act = qt.QAction(’List␣Maps’, self)
        list_entries_act.triggered.connect(self.list_entries)
        database_menu.addAction(list_entries_act)

        menubar.addAction(database_menu.menuAction())

    def configure_window(self):
        self.setWindowTitle(’Augmented␣Maps’)
        screen_size = gui.QGuiApplication.primaryScreen().
              availableSize()
        self.resize(int(screen_size.width() ∗ 3 / 5),
                    int(screen_size.height() ∗ 3 / 5))
        self.center()
        self.statusBar().showMessage(’Ready’)

    def open_camera_calibration(self):
        utils.camera_calibration_matrix()
        return

    def open_capture(self):

        self.scene.clear()
        a = 0

        counter = 0

        # camera calibration matrix
        with open(’camera_parameters.yaml’) as f:
            loadeddict = yaml.load(f)

        mtx = loadeddict.get(’camera_matrix’)
        self.DIST = loadeddict.get(’dist_coeff’)
        mtx = mtx.ravel()
        AugmentedMaps.MTX = [[mtx[0], mtx[1], mtx[2]], [
            mtx[3], mtx[4], mtx[5]], [mtx[6], mtx[7], mtx[8]]]
    kp = None
    goodImages = []
    found_match = False

    video = cv2.VideoCapture(0)

    while True:

        check, frame = video.read()

        if check:

            if counter % 1 == 0:
                found_match, kp, goodImages = self.
                     compute_match(
                    frame, self.database)

                # In order to reduce computer power:
                if (not self.entry == None) and found_match ==
                     False:
                    self.entry = None

            if found_match:
                frame = self.augment_map(
                    kp, goodImages[0][0], frame, goodImages
                         [0][1])
            else:
                frame = cv2.cvtColor(frame, cv2.
                     COLOR_BGR2RGB)

            try:
                counter = counter + 1
            except:
                counter = 0

            # show frame
            self.scene.addPixmap(gui.QPixmap(utils.
                 numpy_to_qimage(frame)))

            #cv2.imshow(’image’, frame)
            key = cv2.waitKey(1)
            if keyboard.is_pressed(’q’):
                break

    video.release()

    self.scene.clear()

def open_image_map(self):
    filename, __ = qt.QFileDialog.getOpenFileName(self, ’Load␣
         Image’, os.environ.get(’HOME’),
                                                  ’Images␣(∗.
                                                      jpg␣∗.
                                                      jpeg␣
                                                      ∗.png)
                                                      ’)
    if filename:
        image = cv2.imread(filename)
        img = image
        found, kp, img, goodImages = self.compute_match(
            image, self.database)
        if True == found:
            image = self.augment_map(
                kp, goodImages[0][0], img, goodImages[0][1])
        else:
            image = cv2.cvtColor(image, cv2.
                 COLOR_BGR2RGB)
        # Draw result in screen
        self.scene.clear()
        self.scene.addPixmap(gui.QPixmap(utils.
             numpy_to_qimage(image)))
        self.update()

def compute_match(self, image, database):
    image_hist_eq = utils.histogram_equalization(image)
    try:
        kp, des = utils.get_features(image_hist_eq)
    except:
        return False, None, None

    goodImages = []
    if self.entry == None:
        for entry in database.entries:
            # if AugmentedMaps.debug:
            print(f"Matching␣features␣with␣{entry.name}")
            self.entry = entry
            matches = utils.match_descriptors(entry.descriptors,
                 des)
            if AugmentedMaps.debug:
                print(f"Found␣{len(matches)}␣descriptor␣
                     matches")

            if len(matches) >= 80:
                print(f"Found␣a␣match:␣{entry.name}")
                goodImages.append((matches, entry))
            #goodImages.append((matches, entry))
    else:
        matches = utils.match_descriptors(self.entry.descriptors,
             des)
        if len(matches) >= 80:
            goodImages.append((matches, self.entry))

    if goodImages == [] or len(goodImages) == 0:
        return False, None, None

    return True, kp, sorted(goodImages, key=lambda x: len(x[0])
         )

@staticmethod
def augment_map(kp, matches, image, image_prepared):
    # Calculates source and destination points
    src_pts = np.float32([image_prepared.keypoints[m.queryIdx][’
         pt’]
                                  for m in matches]).reshape(−1, 1, 2)
    dst_pts = np.float32(
        [kp[m.trainIdx].pt for m in matches]).reshape(−1, 1, 2)
    if AugmentedMaps.debug:
        print(’Calculating␣Homography’)
    # homography
    matrix, _ = cv2.findHomography(src_pts, dst_pts, cv2.
         RANSAC, 5.0)

    # Get width and height from image
    h, w, __ = np.shape(image)

    # Converts color namespace from BGR to RGB
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    # Verifies if the image map has any Point of Interest
    if len(image_prepared.interestPoints) > 0:
        # Gets the nearest Point of Interest from the center
        try:
            if AugmentedMaps.debug:
                print(’Calculating␣nearest␣interesting␣point’)
            nearest_interestpoint = utils.get_nearest_interestpoint(
                image_prepared, matrix, w, h)
        except:
            return image
        # Resize the image of the Point of Interest
        interestImage = cv2.cvtColor(
            nearest_interestpoint[3], cv2.COLOR_BGR2RGB)
        if image.shape[0] > image.shape[1]:
```

```
    interestPointImage = cv2.resize(
        interestImage, (int(0.30*image.shape[1]), int
            (0.25*image.shape[0])), interpolation=cv2.
            INTER_CUBIC)
elif image.shape[0] <= image.shape[1]:
    interestPointImage = cv2.resize(
        interestImage, (int(0.25*image.shape[1]), int
            (0.30*image.shape[0])), interpolation=cv2.
            INTER_CUBIC)
else:
    interestPointImage = cv2.resize(
        interestImage, (int(0.30*image.shape[1]), int
            (0.30*image.shape[0])), interpolation=cv2.
            INTER_CUBIC)

# Calculates the centroid of the Point of Interest image to
    be drawn
interestPointCentroid = utils.get_centroid(
    (nearest_interestpoint[0][0][0], nearest_interestpoint
        [0][1][0], nearest_interestpoint[0][2][0],
        nearest_interestpoint[0][3][0]))

# Verifies the location of the Point of Interest and
    calculates the position of the its image associated to
    be drawn
if interestPointCentroid[0] < w/2:
    interesPointImageXi = w - interestPointImage.shape
        [1]
    interesPointImageYi = h - interestPointImage.shape
        [0]
    interesPointImageXf = w
    interesPointImageYf = h
    interestPointImageCorderX = interesPointImageXi
    interestPointImageCorderY = interesPointImageYi -
        29
else:
    interesPointImageXi = 0
    interesPointImageYi = h - interestPointImage.shape
        [0]
    interesPointImageXf = interestPointImage.shape[1]
    interesPointImageYf = h
    interestPointImageCorderX = interesPointImageXf
    interestPointImageCorderY = interesPointImageYi -
        29

if AugmentedMaps.debug:
    print('Calculating␣projection␣to␣draw␣pyramid')
projection = utils.projection_matrix(AugmentedMaps.
    MTX, matrix)
image = utils.render(image, projection, w/2, h/2)
# Draw image of the Point of Interest in the map
image[interesPointImageYi:interesPointImageYf,
     interesPointImageXi: interesPointImageXf] =
         interestPointImage
if AugmentedMaps.debug:
    print('Drwaing␣interesting␣point␣image')
# Draws an header for the Point of Interest Image
headerPts = utils.get_header_points(
    interesPointImageXi, interesPointImageYi,
        interesPointImageXf)

image = cv2.fillPoly(
    image, [np.int32(headerPts)], (255, 255, 255))

# Draws a line from the header to the center of the nearest
    Point of Interest
cv2.line(image, (int(interestPointCentroid[0]), int(
    interestPointCentroid[1])), (
    int(interestPointImageCorderX), int(
        interestPointImageCorderY)), (255, 255, 255), 2)
```

```
        # Calculates distance between the center and the Point of
            Interest
        scale = image_prepared.scale
        interestPointDistance = int(scale * nearest_interestpoint
            [1])

        interestPointText = nearest_interestpoint[2] + \
            "␣-␣" + str(interestPointDistance) + "␣m"

        # Draw name of the Point of Interest
        cv2.putText(image, interestPointText, (
            int(headerPts[1][0][0] + 5), int(headerPts[1][0][1] -
                10)), cv2.FONT_HERSHEY_SIMPLEX, 0.4, 0)

        # Draws the location of the nearest Point of Interest
        image = cv2.polylines(
            image, [np.int32(nearest_interestpoint[0])], True, 255,
                3, cv2.LINE_AA)

    if AugmentedMaps.debug:
        print('Drawing␣compass')
    # Gets the points of the compass
    pts_compass = utils.get_compass_points(w, h)

    # Project corners into frame
    dst_compass = cv2.perspectiveTransform(pts_compass, matrix)

    wDiff = int(w/2 + 30 - dst_compass[0][0][0])
    hDiff = int(h/2 - dst_compass[0][0][1])

    dst_compass[0][0][0] = dst_compass[0][0][0] + wDiff
    dst_compass[0][0][1] = dst_compass[0][0][1] + hDiff
    dst_compass[1][0][0] = dst_compass[1][0][0] + wDiff
    dst_compass[1][0][1] = dst_compass[1][0][1] + hDiff
    dst_compass[2][0][0] = dst_compass[2][0][0] + wDiff
    dst_compass[2][0][1] = dst_compass[2][0][1] + hDiff
    dst_compass[3][0][0] = dst_compass[3][0][0] + wDiff
    dst_compass[3][0][1] = dst_compass[3][0][1] + hDiff

    # Connect the corners of the compass with lines
    image = cv2.polylines(
        image, [np.int32(dst_compass)], True, 0, 2, cv2.LINE_AA)

    image = cv2.fillPoly(
        image, [np.int32([dst_compass[0], dst_compass[2],
            dst_compass[3]])], (150, 0, 0))

    image = cv2.fillPoly(
        image, [np.int32([dst_compass[0], dst_compass[1],
            dst_compass[2]])], (0, 0, 150))

    # Draws a circle at the center of the map
    image = utils.draw_center_map(image, w, h)

    return image

def open_add_entry_window(self):
    if AugmentedMaps.debug:
        print('Opening␣an␣image␣map')
    self.__entryWindow = Preparation(self.database)
    pos = self.frameGeometry().topLeft()
    self.__entryWindow.move(pos.x() + 20, pos.y() + 20)
    self.__entryWindow.show()

def list_entries(self):
    self.popup_list = EntriesList(self, self.database)

def center(self):
    qr = self.frameGeometry()
    cp = qt.QDesktopWidget().availableGeometry().center()
    qr.moveCenter(cp)
```

```
        self.move(qr.topLeft())

    def closeEvent(self, event):
        reply = qt.QMessageBox.question(self, 'Message',
                                        "Are␣you␣sure␣to␣you␣
                                            want␣to␣quit?", qt.
                                            QMessageBox.Yes |
                                        qt.QMessageBox.No, qt.
                                            QMessageBox.No)

        if reply == qt.QMessageBox.Yes:
            event.accept()
        else:
            event.ignore()


class EntriesList(qt.QWidget):
    class ListEntry(qt.QWidget):
        deleted = qtc.pyqtSignal(qt.QWidget)

        def __init__(self, parent, entry: ImageMap):
            super().__init__(parent)
            self.entry = entry
            layout = qt.QGridLayout()
            image = qt.QLabel()
            image.setPixmap(gui.QPixmap(
                utils.image_to_qimage(entry.img)).scaledToWidth
                    (300))
            layout.addWidget(image, 0, 0, Qt.AlignCenter)
            layout.addWidget(qt.QLabel("%s" %
                                       (entry.name)), 1, 0, Qt.
                                           AlignCenter)
            delete_btn = qt.QPushButton("Delete", self)
            delete_btn.released.connect(lambda: self.deleted.emit(self
                ))
            layout.addWidget(delete_btn, 2, 0, Qt.AlignCenter)
            self.setLayout(layout)

    def __init__(self, parent, database):
        super().__init__(parent)
        self.database = database

        self.area = qt.QScrollArea()
        widget = qt.QWidget()
        self.layout = qt.QVBoxLayout()
        self.layout.setContentsMargins(0, 0, 0, 0)
        for e in self.database.entries:
            list_entry = self.ListEntry(self, e)
            list_entry.deleted.connect(self.delete_entry)
            self.layout.addWidget(list_entry)

        widget.setLayout(self.layout)
        self.area.setWidget(widget)
        self.area.show()

    def delete_entry(self, entry: ListEntry):
        self.database.remove_map(entry.entry)
        self.layout.removeWidget(entry)
        entry.deleteLater()
        self.layout.update()
```

D database.py

```
import pickle
from typing import List, Optional
import numpy as np
from image_map import ImageMap


# Abstraction to save/return image maps prepared
class Database:
    def __init__(self, filename):
        self.filename = filename
        self.__imageMaps = dict()

    @classmethod
    def connect(cls, filename):
        try:
            file = open(filename, 'rb')
        except FileNotFoundError:
            db = cls(filename)
            db.save()
            return db
        else:
            return pickle.load(file)

    # Returns the saved maps
    @property
    def entries(self):
        return self.__imageMaps.values()

    # Returns a map
    def imageMap(self, name) −> Optional[ImageMap]:
        return self.__imageMaps.get(name)

    # Saves the current state of the database
    def save(self):
        with open(self.filename, 'wb') as file:
            print(f'Saving␣data␣to␣database:␣{self.filename}')
            pickle.dump(self, file, pickle.HIGHEST_PROTOCOL)

    # Adds a new map to the database
    def add_map(self, imageMap: ImageMap):
        self.__imageMaps[imageMap.name] = imageMap
        self.save()

    # Removes a map from the database
    def remove_map(self, imageMap: ImageMap):
        if self.__imageMaps[imageMap.name]:
            del self.__imageMaps[imageMap.name]
            self.save()
```

E gui$_e$*ditor.py*

```
from enum import Enum, unique, auto
from typing import List, Set, Tuple

from PyQt5 import (QtWidgets as qt,
                   QtGui as gui,
                   QtCore as qtc)
from PyQt5.QtCore import Qt
import numpy as np
import math
import cv2
from interest_point_augment_graphic import
    InterestPointAugmentGraphic
from image_selection_dialog import ImageDlg

@unique
class EditorState(Enum):
    NONE = auto()
    INSERT_AUGMENT_ITEM = auto()


class EditorScene(qt.QGraphicsScene):
    entry_changed = qtc.pyqtSignal()

    def __init__(self):
        super().__init__()
        self.state: EditorState = EditorState.NONE
        self.entry: dict = None
        self._selection_rect: dict = None
        self._selection_rect_ui: qt.QGraphicsRectItem = None
        self.augments: Set[InterestPointAugmentGraphic] = set()
        self._dragging: InterestPointAugmentGraphic = None
        self._selected: InterestPointAugmentGraphic = None
        self._item_start_point: Tuple[float, float] = None
        self._item: InterestPointAugmentGraphic = None

        self.delete_act = qt.QAction(’Delete␣Point␣of␣Interest’, self)
        self.delete_act.triggered.connect(self.delete_point_of_interest)

    def load_map(self, img):
        self.state = EditorState.NONE
        self._selected = None
        self._dragging = None
        self.clear()

        # Get map dimensions
        h, w, d = np.shape(img)

        # Converts map to RGB
        rgbImg = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

        q_image = gui.QImage(rgbImg, w, h, w ∗ d, gui.QImage.
              Format_RGB888)
        entry_item = self.addPixmap(gui.QPixmap(q_image))
        self.setSceneRect(entry_item.boundingRect())
        self.update()
        self.entry = {’id’: None,
                      ’img’: img,
                      ’gui’: entry_item}
        self.entry_changed.emit()

    def delete_point_of_interest(self):
        if self._selected:
            self.augments.remove(self._selected)
            self.removeItem(self._selected)
            self._selected = None
            self.update()

    def contextMenuEvent(self, event: qt.
          QGraphicsSceneContextMenuEvent):
        item = self.itemAt(event.scenePos(), gui.QTransform())
        item = item if item and isinstance(
            item, InterestPointAugmentGraphic) else self._selected
        if item:
            self._selected = item
            context_menu = qt.QMenu()
            context_menu.addAction(self.delete_act)
            context_menu.exec(event.screenPos())
            event.accept()
        else:
            event.ignore()

    def mousePressEvent(self, event: qt.QGraphicsSceneMouseEvent):
        if event.button() != Qt.LeftButton:
            return

        pos = event.scenePos()

        if self.state is EditorState.INSERT_AUGMENT_ITEM:
            self._item_start_point = (pos.x(), pos.y())
        elif self.state is EditorState.NONE:
            item = self.itemAt(event.scenePos(), gui.QTransform())
            item = item if item and isinstance(
                item, InterestPointAugmentGraphic) else None
            self._dragging = item
            if self._selected and item != self._selected:
                self._selected.selected = False
            self._selected = item
            if self._selected:
                self._selected.selected = True
            self.update()
        event.accept()

    def mouseReleaseEvent(self, event: qt.QGraphicsSceneMouseEvent
          ):
        if self.state is EditorState.INSERT_AUGMENT_ITEM:
            if self._item:
                self._item.drawing = False
                dlg = ImageDlg()
                if dlg.exec():
                    name = dlg.name
                    images = dlg.images
                    self._item.setName(name)
                    self._item.setImages(images)
                else:
                    self._selected = self._item
                    self.delete_point_of_interest()

                    return
                self.augments.add(self._item)
                self._item_start_point = None
                self._item = None
                self.update()
        elif self.state is EditorState.NONE:
            if self._dragging:
                self._dragging.dragging = False
                self._dragging = None
                self.update()

        event.accept()

    def mouseMoveEvent(self, event: qt.QGraphicsSceneMouseEvent):
        if self.state is EditorState.INSERT_AUGMENT_ITEM:
            if self._item_start_point:
                if not self._item:
                    self._item = InterestPointAugmentGraphic(0, 0)
                    self._item.setPos(∗self._item_start_point)
                    self._item.drawing = True
                    self.addItem(self._item)
                box = self._item
                box.width = event.scenePos().x() − self.
                     _item_start_point[0]
                box.height = event.scenePos().y() − self.
                     _item_start_point[1]
                self.update()
        elif self.state is EditorState.NONE and self._dragging is not
             None:
            curr = (event.scenePos().x(), event.scenePos().y())
            prev = (event.lastScenePos().x(), event.lastScenePos().y())
            delta = (curr[0] − prev[0], curr[1] − prev[1])
            self._dragging.dragging = True
            self._dragging.moveBy(∗delta)
            self.update()
        event.accept()


class EditorView(qt.QGraphicsView):
    mouse_moved = qtc.pyqtSignal(object)

    def __init__(self, scene: EditorScene):
        super().__init__(scene)
        self.editor_scene = scene
        scene.entry_changed.connect(self.handle_entry_changed)
        self.viewport().grabGesture(Qt.PinchGesture)
        self.viewport().setMouseTracking(True)
        self.setFrameStyle(0)

    def handle_entry_changed(self):
        self.reset_zoom()
        self.fit_to_entry()

    def fit_to_entry(self):
        if self.editor_scene.entry is not None:
            self.fitInView(self.editor_scene.entry['gui'], Qt.
                KeepAspectRatio)

    def reset_zoom(self):
        self.resetTransform()

    def resizeEvent(self, event: gui.QResizeEvent):
        self.fit_to_entry()
        super().resizeEvent(event)

    def viewportEvent(self, event: qtc.QEvent):
        if event.type() == qtc.QEvent.Gesture:
            return self.gesture_event(event)
        return super().viewportEvent(event)

    def mouseMoveEvent(self, event: gui.QMouseEvent):
        scene_pos = self.mapToScene(event.pos())
        self.mouse_moved.emit((scene_pos.x(), scene_pos.y()))
        super().mouseMoveEvent(event)

    def gesture_event(self, event: qt.QGestureEvent) −> bool:
        pinch: qt.QPinchGesture = event.gesture(Qt.PinchGesture)
        if pinch is not None:
            zoom_factor = pinch.totalScaleFactor()
            self.setTransformationAnchor(qt.QGraphicsView.
                NoAnchor)
            self.setResizeAnchor(qt.QGraphicsView.NoAnchor)
            self.scale(zoom_factor, zoom_factor)
        return True
```

F image$_{m}ap.py$

```
import pickle
from typing import List, Optional
import numpy as np
from interest_point import InterestPoint


class ImageMap:
    def __init__(self, name: str, scale, image, keypoints, descriptors,
            interestPoints: Optional[List[InterestPoint]] = None):
        self.name = name
        self.img = image
        self.keypoints = keypoints
        self.descriptors = descriptors
        self.interestPoints = interestPoints if interestPoints is not None
            else []
        self.scale = scale
```

G image$_{s}election_{d}ialog.py$

```
# −*− coding: utf−8 −*−

# Form implementation generated from reading ui file 'C:\Users\Luis\
    Documents\rvau2.ui'
#
# Created by: PyQt5 UI code generator 5.11.3
#
# WARNING! All changes made in this file will be lost!
import cv2
from PyQt5 import QtCore, QtGui, QtWidgets

class ImageDlg(QtWidgets.QDialog):

    def __init__(self):
        super(ImageDlg, self).__init__()
        self.setupUi(self)
        self.images = []
        self.imageName = set()
        self.name = None

    def setupUi(self, Dialog):

        Dialog.setObjectName("Dialog")
        Dialog.resize(809, 599)
        self.verticalLayoutWidget = QtWidgets.QWidget(Dialog)
        self.verticalLayoutWidget.setGeometry(QtCore.QRect(0, 0,
            811, 601))
        self.verticalLayoutWidget.setObjectName("
            verticalLayoutWidget")
        self.verticalLayout = QtWidgets.QVBoxLayout(self.
            verticalLayoutWidget)
        self.verticalLayout.setContentsMargins(0, 0, 0, 0)
        self.verticalLayout.setObjectName("verticalLayout")
        self.horizontalLayout_3 = QtWidgets.QHBoxLayout()
        self.horizontalLayout_3.setContentsMargins(100, 20, 100, 20)
        self.horizontalLayout_3.setSpacing(6)
        self.horizontalLayout_3.setObjectName("horizontalLayout_3")
        self.label_2 = QtWidgets.QLabel(self.verticalLayoutWidget)
        self.label_2.setMinimumSize(QtCore.QSize(200, 0))
        self.label_2.setMaximumSize(QtCore.QSize(16777215, 50))
        font = QtGui.QFont()
        font.setPointSize(12)
        font.setKerning(True)
        self.label_2.setFont(font)
        self.label_2.setAutoFillBackground(True)
        self.label_2.setFrameShape(QtWidgets.QFrame.Box)
        self.label_2.setAlignment(QtCore.Qt.AlignCenter)
        self.label_2.setObjectName("label_2")
        self.horizontalLayout_3.addWidget(self.label_2)
        self.textEdit = QtWidgets.QTextEdit(self.verticalLayoutWidget
            )
        self.textEdit.setMaximumSize(QtCore.QSize(16777215, 50))
        font = QtGui.QFont()
        font.setPointSize(12)
        self.textEdit.setFont(font)
        self.textEdit.setObjectName("textEdit")
        self.horizontalLayout_3.addWidget(self.textEdit)
        self.verticalLayout.addLayout(self.horizontalLayout_3)
        self.verticalLayout_2 = QtWidgets.QVBoxLayout()
        self.verticalLayout_2.setObjectName("verticalLayout_2")
        self.horizontalLayout_2 = QtWidgets.QHBoxLayout()
        self.horizontalLayout_2.setContentsMargins(50, 50, 50, 50)
        self.horizontalLayout_2.setSpacing(50)
        self.horizontalLayout_2.setObjectName("horizontalLayout_2")
        self.label = QtWidgets.QLabel(self.verticalLayoutWidget)
        self.label.setFrameShape(QtWidgets.QFrame.Box)
        self.label.setText("")
        self.label.setObjectName("label")
        self.horizontalLayout_2.addWidget(self.label)
        self.listWidget = QtWidgets.QListWidget(self.
            verticalLayoutWidget)
        self.listWidget.setFrameShape(QtWidgets.QFrame.StyledPanel
            )
        self.listWidget.setAlternatingRowColors(False)
        self.listWidget.setObjectName("listWidget")
        self.horizontalLayout_2.addWidget(self.listWidget)
        self.horizontalLayout_2.setStretch(0, 3)
        self.horizontalLayout_2.setStretch(1, 1)
        self.verticalLayout_2.addLayout(self.horizontalLayout_2)
        self.verticalLayout.addLayout(self.verticalLayout_2)
        self.verticalLayout_3 = QtWidgets.QVBoxLayout()
        self.verticalLayout_3.setObjectName("verticalLayout_3")
        self.horizontalLayout = QtWidgets.QHBoxLayout()
        self.horizontalLayout.setContentsMargins(50, −1, 50, 10)
        self.horizontalLayout.setSpacing(50)
        self.horizontalLayout.setObjectName("horizontalLayout")
        self.AddImage = QtWidgets.QPushButton(self.
            verticalLayoutWidget)
        self.AddImage.setMinimumSize(QtCore.QSize(0, 40))
        font = QtGui.QFont()
        font.setPointSize(12)
        self.AddImage.setFont(font)
        self.AddImage.setObjectName("AddImage")
        self.horizontalLayout.addWidget(self.AddImage)
        self.pushButton = QtWidgets.QPushButton(self.
            verticalLayoutWidget)
        self.pushButton.setMinimumSize(QtCore.QSize(0, 40))
        font = QtGui.QFont()
        font.setPointSize(12)
        self.pushButton.setFont(font)
        self.pushButton.setObjectName("pushButton")
        self.horizontalLayout.addWidget(self.pushButton)
        self.deleteImage = QtWidgets.QPushButton(self.
            verticalLayoutWidget)
        self.deleteImage.setMinimumSize(QtCore.QSize(0, 40))
        font = QtGui.QFont()
        font.setPointSize(12)
        self.deleteImage.setFont(font)
        self.deleteImage.setObjectName("deleteImage")
        self.horizontalLayout.addWidget(self.deleteImage)
        self.verticalLayout_3.addLayout(self.horizontalLayout)
        self.verticalLayout.addLayout(self.verticalLayout_3)
        self.verticalLayout.setStretch(0, 1)
        self.verticalLayout.setStretch(1, 5)
        self.verticalLayout.setStretch(2, 1)

        self.retranslateUi(Dialog)
        self.listWidget.setCurrentRow(−1)
        QtCore.QMetaObject.connectSlotsByName(Dialog)

        self.pushButton.clicked.connect(self.finish)
        self.AddImage.clicked.connect(self.addImage)
        self.deleteImage.clicked.connect(self.delete)



    def retranslateUi(self, Dialog):
        _translate = QtCore.QCoreApplication.translate
        Dialog.setWindowTitle(_translate("Dialog", "Dialog"))
        self.label_2.setText(_translate("Dialog", "Interest␣Point␣Name
            :"))
        self.AddImage.setText(_translate("Dialog", "Add␣Image"))
        self.pushButton.setText(_translate("Dialog", "Finish"))
        self.deleteImage.setText(_translate("Dialog", "Delete␣Last␣
            Image"))


    def addImage(self):
        fileName, _ = QtWidgets.QFileDialog.getOpenFileName(None,
             "Select␣Image", "", "Image␣Files␣(∗.png␣∗.jpg␣∗.jpeg)
            ")
        if fileName and fileName not in self.imageName:
            img = cv2.imread(fileName)

            pixmap = QtGui.QPixmap(fileName)
            pixmap = pixmap.scaled(self.label.width(), self.label.
                height(), QtCore.Qt.KeepAspectRatio )
            self.label.setPixmap(pixmap)
            self.label.setAlignment(QtCore.Qt.AlignCenter)
            array = fileName.split('/')
            self.listWidget.addItem(array [len(array)−1 ])
            self.imageName.add(fileName)

            self.images.append(img)

    def delete(self):
        #self.label.setPixmap(None)
        size = self.listWidget.count()
        itemP = self.listWidget.item(size−1)
        self.listWidget.takeItem(size−1)
        self.imageName.pop()
        self.images.pop()

    def finish(self):
        self.name = self.textEdit.toPlainText()
        if not self.name or len(self.images) < 1:
            info_box = QtWidgets.QMessageBox(self)
            info_box.setIcon(QtWidgets.QMessageBox.Critical)
            info_box.setText("Name␣can't␣be␣empty␣and␣you␣must
                ␣pick␣at␣least␣one␣image!")
            return info_box.exec()
        else:
            self.accept()
```

H interest$_{p}oint.py$

```
# Class that abstracts a point of interest in the map
class InterestPoint():
    def __init__(self, x, y, w, h):
        super().__init__()
        self.x = x
        self.y = y
        self.w = w
        self.h = h
        self.name = None
        self.images = []

    def setName(self, name):
        self.name = name

    def setImage(self, image):
        self.images = image
```

I interest$_{p}oint_{a}ugment_{g}raphic.py$

```
from abc import ABC
from typing import Optional
from PyQt5 import (QtWidgets as qt,
                   QtGui as gui,
                   QtCore as qtc)
from PyQt5.QtCore import Qt
import math
from interest_point import InterestPoint


class InterestPointAugmentGraphic(qt.QGraphicsItem):
    def __init__(self, width: float = 100, height: float = 100):
        super().__init__()
        self._dragging: bool = False
        self._selected: bool = False
        self._drawing: bool = False
        self.setCursor(Qt.PointingHandCursor)
        self._width: float = width
        self._height: float = height
        self._name = None
        self._image = None

    @property
    def dragging(self) −> bool:
        return self._dragging

    @dragging.setter
    def dragging(self, value: bool):
        self.setCursor(Qt.ClosedHandCursor if value else Qt.
            PointingHandCursor)
        self._dragging = value

    @property
    def selected(self) −> bool:
        return self._selected
    @selected.setter
    def selected(self, value: bool):
        self._selected = value

    @property
    def drawing(self) -> bool:
        return self._drawing

    @drawing.setter
    def drawing(self, value: bool):
        self._drawing = value

    @property
    def width(self):
        return self._width

    @width.setter
    def width(self, width: float):
        self.prepareGeometryChange()
        self._width = width

    @property
    def height(self):
        return self._height

    @height.setter
    def height(self, height: float):
        self.prepareGeometryChange()
        self._height = height

    def setName(self, name):
        self._name = name

    def setImages(self, imgs):
        self._image = imgs

    def boundingRect(self) -> qtc.QRectF:
        return qtc.QRectF(0, 0, self.width + 5, self.height + 5)

    def paint(self, painter: gui.QPainter, option: qt.
         QStyleOptionGraphicsItem, widget: Optional[qt.QWidget] =
         ...):
        color = gui.QColor(255, 0, 0, 255)
        if self.dragging:
            color.setAlphaF(0.7)
        if self._selected:
            color.setGreen(200)
        if self._drawing:
            color = gui.QColor(120, 32, 32, 100)
        pen = gui.QPen()
        pen.setColor(color)
        pen.setWidth(5)
        painter.setPen(pen)
        painter.drawRect(2, 2, int(self.width), int(self.height))

    def getInterestPoint(self):
        interestPoint = InterestPoint(
            self.x(), self.y(), self.width, self.height)
        interestPoint.setName(self._name)
        interestPoint.setImage(self._image)

        return interestPoint
```

J preparation.py

```
import os
import cv2
from typing import Optional, Tuple
from PyQt5 import (QtWidgets as qt,
                   QtGui as gui,
                   QtCore as qtc)
from PyQt5.QtCore import Qt
from image_map import ImageMap
from database import Database
from gui_editor import EditorScene, EditorState, EditorView
import utils


class Preparation(qt.QMainWindow):
    entry_saved = qtc.pyqtSignal(ImageMap)

    def __init__(self, database: Database):
        super().__init__()
        self._database = database
        self.img = None
        self.toolbar: qt.QToolBar = None
        self.tool_save: qt.QAction = None
        self.configure_window()
        self.configure_toolbar()

        self.editor_scene = EditorScene()
        self.editor_scene.entry_changed.connect(self.on_entry_change)
        self.editor_view = EditorView(self.editor_scene)
        self.editor_view.mouse_moved.connect(self.on_mouse_move)
        self.entry_name_combo: qt.QComboBox = None
        self.sidebar = self.create_sidebar()

        splitter = qt.QSplitter(Qt.Horizontal, self)

        splitter.addWidget(self.editor_view)
        splitter.addWidget(self.sidebar)
        splitter.setStretchFactor(0, 1)
        splitter.setStretchFactor(1, 0)

        self.setCentralWidget(splitter)




    def create_sidebar(self) -> qt.QWidget:
        sidebar = qt.QWidget()
        layout = qt.QVBoxLayout()
        layout.setContentsMargins(0, 0, 0, 0)
        area = qt.QScrollArea()
        area.setFrameStyle(0)
        content = qt.QWidget()
        content_layout = qt.QVBoxLayout()
        content_layout.setContentsMargins(0, 0, 0, 8)

        info_box = qt.QWidget()
        form = qt.QFormLayout()
        self.entry_name_combo = qt.QComboBox(info_box)
        self.entry_name_combo.setEditable(True)
        self.entry_name_combo.setEditText(’’)

        self.verticalLayoutWidget = qt.QWidget()
        self.verticalLayoutWidget.setGeometry(qtc.QRect(410, 120,
             181, 161))
        self.verticalLayoutWidget.setObjectName("
             verticalLayoutWidget")
        self.verticalLayout = qt.QVBoxLayout(self.
             verticalLayoutWidget)
        self.verticalLayout.setContentsMargins(0, 0, 0, 0)
        self.verticalLayout.setSpacing(0)
        self.verticalLayout.setObjectName("verticalLayout")
        self.label = qt.QLabel(self.verticalLayoutWidget)
        self.label.setMaximumSize(qtc.QSize(16777215, 40))
        font = gui.QFont()
        font.setPointSize(10)
        self.label.setFont(font)
        self.label.setAutoFillBackground(True)
        self.label.setFrameShape(qt.QFrame.Box)
        self.label.setObjectName("label")
        self.verticalLayout.addWidget(self.label)
        self.Name = qt.QTextEdit(self.verticalLayoutWidget)
        self.Name.setMaximumSize(qtc.QSize(16777215, 30))
        font = gui.QFont()
        font.setPointSize(10)
        self.Name.setFont(font)
        self.Name.setObjectName("Name")
        self.verticalLayout.addWidget(self.Name)
        self.label_2 = qt.QLabel(self.verticalLayoutWidget)
        self.label_2.setMaximumSize(qtc.QSize(16777215, 40))
        font = gui.QFont()
        font.setPointSize(10)
        self.label_2.setFont(font)
        self.label_2.setAutoFillBackground(True)
        self.label_2.setFrameShape(qt.QFrame.Box)
        self.label_2.setObjectName("label_2")
        self.verticalLayout.addWidget(self.label_2)
        self.Name_2 = qt.QTextEdit(self.verticalLayoutWidget)
        self.Name_2.setMaximumSize(qtc.QSize(16777215, 30))
        font = gui.QFont()
        font.setPointSize(10)
        self.Name_2.setFont(font)
        self.Name_2.setObjectName("Name_2")
        self.verticalLayout.addWidget(self.Name_2)

        info_box.setLayout(self.verticalLayout)
        content_layout.addWidget(info_box)
        self.augments_group = qt.QButtonGroup(self)
        self.augments_group.setExclusive(False)
        self.augments_group.buttonClicked[int].connect(self.
            augment_clicked)

        augments_widget = qt.QWidget(sidebar)
        augments_layout = qt.QGridLayout()

        box_augment_widget, box_augment_button = self.
            toolbox_interestpoint()
        self.augments_group.addButton(
            box_augment_button, 1)
        augments_layout.addWidget(box_augment_widget, 0, 0)

        augments_widget.setLayout(augments_layout)

        toolbox = qt.QToolBox(sidebar)
        toolbox.addItem(augments_widget, "Points␣of␣Interest")
        content_layout.addWidget(toolbox)

        content_layout.setSizeConstraint(qt.QLayout.SetMinimumSize
            )
        content.setLayout(content_layout)
        area.setWidget(content)
        layout.addWidget(area)
        sidebar.setLayout(layout)

        self.label.setText("Name:")
        self.label_2.setText("Scale:")

        return sidebar

    # Configures window where the image map will be rendered
    def configure_window(self):
        self.setWindowTitle('New␣Map')
        screen_size = gui.QGuiApplication.primaryScreen().
            availableSize()
        self.resize(int(screen_size.width() ∗ 3 / 5),
                    int(screen_size.height() ∗ 3 / 5))
        self.grabGesture(qtc.Qt.PinchGesture)
        self.statusBar().showMessage("Load␣a␣map␣to␣start")

    # Configures toolbar
```

```
    def configure_toolbar(self):
        self.toolbar = self.addToolBar('Main␣Toolbar')

        load_act = self.toolbar_button('Load␣Image')
        load_act.triggered.connect(self.load_image)
        self.toolbar.addAction(load_act)

        self.tool_save = self.toolbar_button('Save␣Map')
        self.tool_save.setDisabled(True)
        self.tool_save.triggered.connect(self.save_map)
        self.toolbar.addAction(self.tool_save)

    # Loads a image map
    def load_image(self):
        filename, __ = qt.QFileDialog.getOpenFileName(self, 'Load␣
            Image', os.environ.get('HOME'),
                                                                  'Images␣(∗.
                                                                     jpg␣∗.
                                                                     jpeg␣
                                                                     ∗.png)
                                                                     ')
        if filename:
            print(f'Loading␣{filename}')
            self.img = cv2.imread(filename)
            self.editor_scene.load_map(self.img)


    # Saves the prepared map in database
    def save_map(self):
        name = self.Name.toPlainText()

        # Map prepared must have a name
        if not name:
            info_box = qt.QMessageBox(self)
            info_box.setIcon(qt.QMessageBox.Critical)
            info_box.setText("Name␣can't␣be␣empty!")
            return info_box.exec()

        # Computes keypoints and descriptors of the image map
        sift = cv2.xfeatures2d.SIFT_create()
        image_hist_eq = utils.histogram_equalization(self.img)
        kp, des = sift.detectAndCompute(image_hist_eq, None)

        # Because pickling cv2.KeyPoint causes PicklingError, we need
             to create a new abstraction for it
        keypoints = utils.keypoints_to_kpdict(kp)

        interestPoints = [a.getInterestPoint()
                          for a in self.editor_scene.augments]

        try:
            scale = int(self.Name_2.toPlainText()); # TODO
        except:
            info_box = qt.QMessageBox(self)
            info_box.setIcon(qt.QMessageBox.Critical)
            info_box.setText("Please␣insert␣a␣valid␣number␣as␣
                scale!")
            return info_box.exec()

        # Creates a new ImageMap with the data from the manipulated
            image
        imageMap = ImageMap(name, scale, self.editor_scene.entry['
            img'], keypoints, des,
                            interestPoints)
        self._database.add_map(imageMap)

        info_box = qt.QMessageBox(self)
        info_box.setIcon(qt.QMessageBox.Information)
        info_box.setText("Saved␣successfully␣as␣'%s'" % imageMap.
            name)
        info_box.exec()
    self.entry_saved.emit(imageMap)

def toolbar_button(self, text: str) −> qt.QAction:
    action = qt.QAction(text, self)
    return action

def toolbox_interestpoint(self) −> Tuple[qt.QWidget, qt.
     QToolButton]:
    button = qt.QToolButton()
    button.setText("Mark␣point␣of␣interest")
    button.setCheckable(True)
    button.setMinimumSize(60, 60)
    grid = qt.QGridLayout()
    grid.addWidget(button, 0, 0, Qt.AlignCenter)
    widget = qt.QWidget()
    widget.setLayout(grid)
    return widget, button

def augment_clicked(self, id: int):
    clicked = self.augments_group.button(id)
    for button in self.augments_group.buttons():
        if clicked != button:
            button.setChecked(False)

    if clicked.isChecked():
        print("Creating␣a␣Point␣of␣Interest")
        self.editor_scene.state = EditorState.
             INSERT_AUGMENT_ITEM
    else:
        self.editor_scene.state = EditorState.NONE

def on_entry_change(self):
    self.tool_save.setDisabled(self.editor_scene.entry is None)

def on_mouse_move(self, scene_position: Tuple[float, float]):
    if self.editor_scene.entry is None:
        return
    status = '(x:␣{:d},␣y:␣{:d})'.format(
        int(scene_position[0]), int(scene_position[1]))
    self.statusBar().showMessage(status)

def closeEvent(self, event):
    reply = qt.QMessageBox.question(self, 'Message',
                                    "You␣haven't␣saved␣the␣
                                         entry␣yet!<br>"
                                    "Are␣you␣sure␣you␣want
                                         ␣to␣close?", qt.
                                         QMessageBox.Yes |
                                    qt.QMessageBox.No, qt.
                                         QMessageBox.No)
    if reply == qt.QMessageBox.Yes:
        event.accept()
    else:
        event.ignore()
```