Realidade Virtual e Aumentada

Mestrado Integrado de Engenharia Informática e Computação

U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Professores
Jorge Alves da Silva
António Augusto de Sousa

Grupo 6:

Inês Caldas - 200904082
Joel Carneiro - 201100775

Índice

Introdução **3**

Descrição da Aplicação **4**

Resultados **5**

Conclusão **6**

Referências **7**

Anexo 1 - GUI 8

Anexo 2 - Código fonte 9

Vision Code 9

GUI Code 26

Introdução

A Realidade Aumentada pode ser definida como uma vista aumentada do mundo físico através de elementos gerados virtualmente. *MonumentAR* é uma aplicação de realidade aumentada que "aumenta" imagens de fachadas de edifícios com elementos visuais. O reconhecimento das fachadas é feito recorrendo ao uso de marcas naturais, não necessitando de marcas fiduciais. A aplicação *MonumentAR* integra dois subprogramas, um de preparação e outro de aumento, que são abstraídos de forma a haver uma melhor interação com o utilizador. Desta forma, o utilizador não tem de navegar por janelas para concluir o seu trabalho.

O subprograma de preparação permite ao utilizador escolher as regiões das imagens que quer usar para calcular os *keypoints,* desenhar os elementos de aumento sobre as imagens da base de dados, para que estes apareçam sobrepostos nas imagens de teste no subprograma de "aumento".

O subprograma de "aumento" tem como função fazer a comparação entre as imagens da base de dados e a imagem escolhida como teste, calcular a matriz de homografia e apresentar os resultados obtidos.

A aplicação *MonumentAR* foi desenvolvida usando a linguagem de programação *Python* com recurso à biblioteca *OpenCV* para efeitos de "aumento" e à biblioteca *tkinter* para a interface gráfica.

O programa mostrou-se capaz de resolver os problemas de teste com eficiência, contudo a sua velocidade depende proporcionalmente dos algoritmos escolhidos pelo utilizador e do tamanho da base de dados.

Nas próximas secções do relatório será apresentada primeiramente uma descrição mais detalhada da aplicação, seguindo-se os resultados obtidos, a análise dos mesmos e as conclusões retiradas.

Descrição da Aplicação

MonumentAR foi desenvolvida usando a linguagem de programação *Python* com recurso à biblioteca *OpenCV* para efeitos de "aumento" e à biblioteca *tkinter* para a interface gráfica.

Após o utilizador instalar todas as dependências da aplicação e abrir o *script* de execução da mesma, é apresentada uma janela onde o utilizador pode efetuar todas as tarefas necessárias.

O utilizador deverá escolher as imagens que quer utilizar como base de dados. Após este passo o utilizador pode fazer o carregamento de uma ou mais imagens para a base de dados e pintar sobre elas as "marcas" que quer apresentar nas futuras imagens a reconhecer. São fornecidas várias ferramentas de edição de imagem, de forma a que o utilizador tenha liberdade na construção de "marcas". Após a criação das mesmas o utilizador deverá guardar as "marcas". Deverá também carregar na opção *Key Points* de forma a escolher qual a região da imagem a ser usada para o cálculos dos pontos de interesse da imagem e remover os *keypoints* que acha desnecessários, caso não o faça a totalidade da imagem é considerada para o cálculo dos mesmos. Poderá efetuar o mesmo processo para as restantes imagens que quiser colocar como base de dados. Pode também aceder à base de dados e eliminar ou alterar as marcas já feitas sobre uma das imagens.

Se a base de dados não se encontrar vazia, o utilizador pode testar diferentes imagens de fachadas de edifícios para visualizar o "aumento" sobre as mesmas de acordo com o conteúdo da base de dados. Para tal, tem à sua disposição dois algoritmos, *Scale-Invariant Feature Transform* (*SIFT*) *[1]* ou *Speeded-Up Robust Features* (*SURF*) *[2]* e o valor a atribuir ao *threshold* do *RANSAC*. O utilizador pode ainda escolher o modo debug onde todos os passos intermédios são apresentados na consola e nos gráficos.

Quanto à utilização da biblioteca *OpenCV*, foram utilizados os seguintes algoritmos, para a computação de *keypoints*, *descriptors* e *matches*:

- SIFT
- SURF
- RANSAC [3] - Random sample consensus
- FLANN [4] - Fast Library for Approximate Nearest Neighbors

Resultados

Foram usadas diferentes imagens, tanto na base de dados, como nas imagens de teste, de forma a testar a aplicação desenvolvida, com o objetivo de verificar o tempo de execução, se a matriz de homografia é calculada corretamente, e que tipo de influências afeta mais cada algoritmo. Os testes foram feitos usando os algoritmo *SIFT* e *SURF* com diferentes valores atribuídos ao limite do algoritmo RANSAC.

Verificou-se que o algoritmo *SIFT* é o mais poderoso. Este mostrou sempre uma boa solução mesmo quando comparando imagens com diferentes iluminações e poses pouco frontais.

Quanto ao algoritmo *SURF*, comprovou-se que, embora seja mais rápido em tempo de execução do que o *SIFT*, o cálculo da homografia, quando se trata de imagens com diferentes iluminações e poses pouco frontais, dá resultados que se podem classificar como maus. É de realçar que a homografia que produziu melhores resultados tendo em conta os *keypoints* e *descriptors* do algoritmo *SURF,* foi quando a imagem da base de dados e a de teste eram a mesma. Desta forma, pode-se concluir que com o algoritmo *SURF* é possível calcular homografias corretas, utilizando os seus *keypoints* e *descriptors* para calcular os *matches*, contudo, os resultados são susceptíveis a grandes variações dependendo das condições de iluminação e pose das imagens.

Para calcular os *matches* foi utilizado o método *FlannBasedMatcher* da biblioteca *FLANN* . Este método é bastante eficiente mas os *matches* são gerados com apenas o *approximate nearest neighbor,* pelo que podem não ser os melhores. Para colmatar esta restrição, aplicou-se sobre os *matches* gerados o algoritmo *RANSAC.* Este mostrou-se um bom ajudante no cálculo de homografias, filtrando bons *matches* dos incorrectos e eliminando os últimos. Quanto menor o valor do limite do *RANSAC* mais *matches* errados são eliminados, resultando um conjunto de *matches* o mais corretos possível para calcular a homografia, dado que este valor representa o limite máximo de variação permitido.

Conclusão

MonumentAR é uma aplicação que visa “aumentar” imagens de fachadas de monumentos, contudo, outras imagens com zonas coplanares podem ser usadas. Esta aplicação foi desenvolvida usando a linguagem de programação *Python* com recurso à biblioteca *OpenCV* para efeitos de “aumento” e à biblioteca *tkinter* para a interface gráfica.

A aplicação permite escolher o tipo de marcas a colocar sobre as imagens, as imagens de base de dados, as imagens de teste e os algoritmos a utilizar. Também permite a escolha da região da imagem da base de dados que irá ser usada para calcular os *keypoints* e os *descriptors.*

Com o trabalho desenvolvido foi possível analisar o resultado do uso dos diferentes tipos de algoritmos de visão por computador, com aplicação na realidade aumentada. Desta forma, e tendo em conta os resultados obtidos, pode-se afirmar que o algoritmo *SIFT* é o algoritmo mais poderoso e que retorna melhores resultados no cálculo de *matches* e homografias com base nos seus *keypoints* e *descriptors*. O *SURF,* por sua vez, é um algoritmo que é muito influenciado por variações de intensidade da luz e variações de perspectiva, o que é uma barreira para o objetivo da aplicação. Tanto o *RANSAC,* como o *FLANN* mostraram ser capazes de otimizar os resultados, especialmente o algoritmo *RANSAC.*

No que diz respeito a trabalho futuro, a aplicação *MonumentAR* poderia ser melhorada a nível de interface gráfica (IG). O trabalho desenvolvido até aqui, embora apresente bons resultados a nível de IG, tem algumas limitações resultantes da pouca experiência com a linguagem *Python* e das limitações da biblioteca *tkinter*. Quanto à parte de visão por computador, poderiam ser adicionados mais algoritmos de forma a que o utilizador tenha mais escolha. Contudo, dois dos principais já são disponibilizados e o *SIFT* apresenta muito bons resultados.

Referências

[1] - OpenCV: Introduction to SIFT (Scale-Invariant Feature Transform). (n.d.). Retrieved December 19, 2017, from https://docs.opencv.org/3.1.0/da/df5/tutorial_py_sift_intro.html

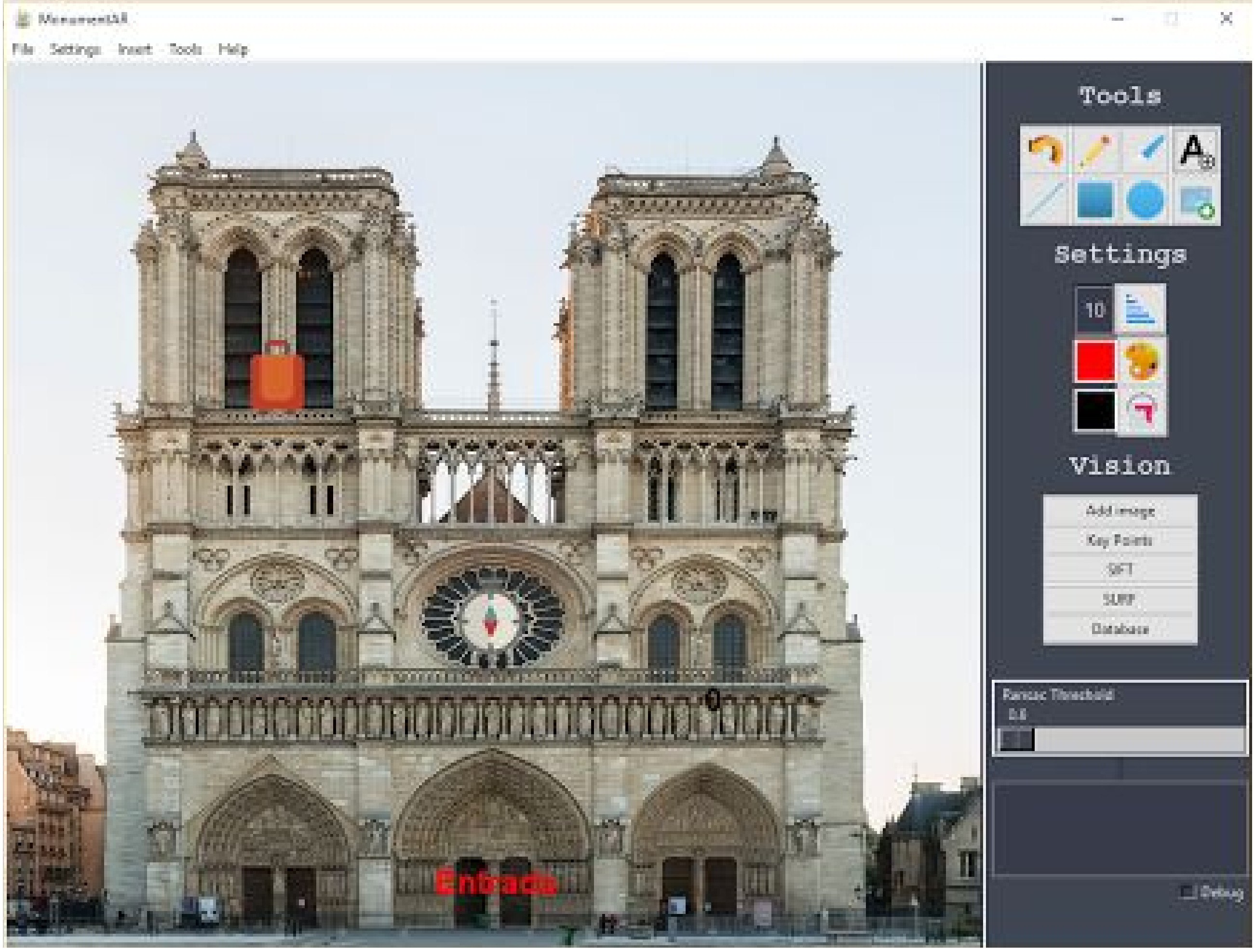
[2] - Introduction to SURF (Speeded-Up Robust Features) — OpenCV 3.0.0-dev documentation. (n.d.). Retrieved December 19, 2017, from https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_surf_intro/py_surf_intro.html

[3] - Feature Matching + Homography to find Objects — OpenCV 3.0.0-dev documentation. (n.d.). Retrieved December 19, 2017, from https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_feature_homography/py_feature_homography.html

[4] - Feature Matching — OpenCV 3.0.0-dev documentation. (n.d.). Retrieved December 19, 2017, from https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_matcher/py_matcher.html

Anexos

Anexo 1 - GUI



Ransac Threshold
0.8

Anexo 2 - Código fonte

Vision Code

```
from __future__ import print_function
from PIL import Image
from vision.database import load_fileImages_database
from vision.feature_points import calculate_feature_points, calculate_matches,
compute_matches, compute_homography
from vision.utils import get_image_layerAR


IMAGE_TEST_PATH = '..\database\sample\image.jpg'

# Tests the user image with databe and computes the homography, then it shows
the results
# Need algorithm type and ransac value
def image_test(image_test, database_images, algorithm_type, ransac_value,
debug_bool):

    #Calculates feature points for test image
        img, kp, des = calculate_feature_points(image_test, algorithm_type,
debug_bool)
    image = [img, kp, des]

    #Calculates matches of image test with all images from database
    matches = calculate_matches(des, database_images[2])

    #If no good match with any of the database images return
    if(len(matches) == 0):
        print('No corresponde with database image was found.', flush = True)
        return

    #Choose database image with best match
    max_matches = 0
    index_max = -1
    for i in range(len(matches)):
        if(len(matches[i]) > max_matches):
            max_matches = len(matches[i])
            index_max = i

    print('Found %d matches for database image %d' % (max_matches, index_max +
 1), flush=True)

    layerAR = get_image_layerAR(index_max)

    database_image = [database_images[0][index_max],
 database_images[1][index_max], database_images[2][index_max]]

    compute_homography(image_test, image, database_image, layerAR,
 matches[index_max], ransac_value, debug_bool)

 # compute the test image and show the findings
 # args : the image path, the algorithm name ('sift' or 'surf'), the ransac
 value (float) and debug (true or false)
 # arAppCompute('..\database\images\img1.jpg', 'surf', 0.6, False)
 def arAppCompute(image_test_path, algorithm_type, ransac_value, debug_bool):
    images_cv, feature_points, descriptors =
 load_fileImages_database(algorithm_type)
    database_images = [images_cv, feature_points, descriptors]

     image_test(image_test_path, database_images, algorithm_type, ransac_value,
 debug_bool)




 import numpy as np
 import cv2
 from time import time
 from matplotlib import pyplot as plt
 from vision.utils import blend_transparent

 # Minimum number of matches
 MIN_MATCH_COUNT = 15
```

```python
# Computes matches for des1 (test image) to des2 (database) using flann based matcher
def compute_matches(des1, des2):
    FLANN_INDEX_KDTREE = 0
    index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
    search_params = dict(checks = 50)
    flann = cv2.FlannBasedMatcher(index_params, search_params)
    matches = flann.knnMatch(des1,des2,k=2)

    good = []
    for m,n in matches:
        if m.distance < 0.7*n.distance:
            good.append(m)

    return good

# Computes the mathes between the test image and the database images
def calculate_matches(image_des, database_des):
    print("Calculating matches with the database images", flush=True)
    start_time = time()
    matches = []
    for db_des in database_des:
        mat = compute_matches(db_des, image_des)
        matches.append(mat)

    end_time = time()
    time_taken = end_time - start_time # time_taken is in seconds
    print("Time spent: %.2f seconds" % time_taken)
    return matches

# Computes the homography between the test image and the image with the higher number of best matches
# Shows the findings with the correct homography
def compute_homography(test_image_path, test_image, database_image, layerAR, matches,ransac_value,debug_bool):

    #kp_database_image/test_image = [img, kp, des]
    layerAR_img = cv2.imread(layerAR, 0)
    coloredLayerAr = cv2.imread(layerAR, -1)
    dst_rgb = cv2.imread(test_image_path, 1)
```

```
   #Images openCV
   src = database_image[0]
   dst = test_image[0]

   #Keypoints
   kp_database_image = database_image[1]
   kp_test_image = test_image[1]

   #Descriptors
   des_database_image = database_image[2]
   des_test_image = test_image[2]

   if len(matches) > MIN_MATCH_COUNT:
           src_pts = np.float32([ kp_database_image[m.queryIdx].pt for m in
matches ]).reshape(-1,1,2)
         dst_pts = np.float32([ kp_test_image[m.trainIdx].pt for m in matches
]).reshape(-1,1,2)

               M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC,
ransac_value)
       matchesMask = mask.ravel().tolist()

       h,w = dst.shape

       result = cv2.warpPerspective(coloredLayerAr, M,(w,h))

       # show findings
       if debug_bool:
           print(layerAR_img.shape, flush=True)
           print(src.shape, flush=True)

       merge = blend_transparent(src, coloredLayerAr)
       merge_final = blend_transparent(dst_rgb, result)

       if debug_bool:
           cv2.namedWindow('res', cv2.WINDOW_KEEPRATIO)
           cv2.resizeWindow('res', 300, 300)
           cv2.imshow('res',result)
           cv2.namedWindow('ori', cv2.WINDOW_KEEPRATIO)
           cv2.resizeWindow('ori', 300, 300)
           cv2.imshow('ori', dst)

           cv2.namedWindow('test', cv2.WINDOW_KEEPRATIO)
           cv2.resizeWindow('test', 300, 300)
           cv2.imshow('test', src)

           cv2.namedWindow('layer', cv2.WINDOW_KEEPRATIO)
           cv2.resizeWindow('layer', 300, 300)
           cv2.imshow('layer',layerAR_img)

       cv2.namedWindow('merge', cv2.WINDOW_KEEPRATIO)
       cv2.resizeWindow('merge', 300, 300)
       cv2.imshow('merge',merge_final)

       cv2.namedWindow('merge_ori', cv2.WINDOW_KEEPRATIO)
       cv2.resizeWindow('merge_ori', 300, 300)
       cv2.imshow('merge_ori',merge)

       # Draw best matches on the screen
          draw_params = dict(matchColor = (0,255,0), # draw matches in green color
                       singlePointColor = None,
                       matchesMask = matchesMask, # draw only inliers
                       flags = 2)

                                                                 img3 = cv2.drawMatches(src,kp_database_image,dst,kp_test_image,matches,None,**draw_params)

       plt.imshow(img3, 'gray'),plt.show()
       cv2.destroyAllWindows()

   else:
         print('The minimum of %s matches was not reached. Please try it agin later...' % MIN_MATCH_COUNT, flush=True)

# calculates the keypoints of an image using a certain algorithm
def calculate_feature_points(image_path, algorithm_type, debug_bool):
   print('Calculating feature points for image %s' % image_path, flush=True)

   if algorithm_type == 'sift':
       print('sift algorithm',flush=True)

       img = cv2.imread(image_path,cv2.IMREAD_GRAYSCALE) # queryImage
       # Initiate SIFT detector
       sift = cv2.xfeatures2d.SIFT_create()

       # find the keypoints and descriptors with SIFT
       kp1, des1 = sift.detectAndCompute(img,None)

       if debug_bool:
           print('kp %s desc %s ' % (len(kp1),len(des1)) ,flush=True)

       return img, kp1, des1

   elif algorithm_type == 'surf':
       print('surf algorithm',flush=True)

       img = cv2.imread(image_path,cv2.IMREAD_GRAYSCALE) # queryImage
       # Create SURF object. You can specify params here or later.
       # Here I set Hessian Threshold to 400
       if debug_bool:
           print('Hessian Threshold = 400',flush=True)

       surf = cv2.xfeatures2d.SURF_create(400)
       """
            Hessian threshold is related to the number of keypoints and descriptores found
          The higher the Hessian threshold value the lower the number of key points and
       descriptors calculated.
       With the chosen value are found around 1700 with the chosen test image
       """
       # Find keypoints and descriptors directly
       kp, des = surf.detectAndCompute(img,None)
       if debug_bool:
           print('kp %s desc %s ' % (len(kp),len(des)) ,flush=True)
        return img, kp, des



from PIL import Image
import glob
from vision.feature_points import calculate_feature_points
import pickle
import os.path
import errno
from os.path import splitext, basename
import numpy as np
import os
from vision.utils import pickle_keypoints, unpickle_keypoints



DATABASE_PATH_IMAGES = '..\\database\\images\\'
DATABASE_PATH_IMAGES_LAYERS = '..\\database\\layers\\'
IMAGES_PATH = '..\database\images_cv'

#SIFT
FILE_PATH_KEYPOINTS_SIFT = '..\\database\\vision\\sift\\keypoints\\'
FILE_PATH_DESCRIPTORS_SIFT = '..\\database\\vision\\sift\\descriptors\\'
FILE_PATH_IMAGE_SIFT = '..\\database\\vision\\sift\\images\\'
FILE_PATH_LOAD_KEYPOINTS_SIFT = "..\\database\\vision\\sift\\keypoints\\*"
FILE_PATH_LOAD_DESCRIPTORS_SIFT = "..\\database\\vision\\sift\\descriptors\\*"
FILE_PATH_IMAGE_LOAD_SIFT = "..\\database\\vision\\sift\\images\\*"

#SURF
FILE_PATH_KEYPOINTS_SURF = '..\\database\\vision\\surf\\keypoints\\'
FILE_PATH_DESCRIPTORS_SURF = '..\\database\\vision\\surf\\descriptors\\'
FILE_PATH_IMAGE_SURF = '..\\database\\vision\\surf\\images\\'
FILE_PATH_LOAD_KEYPOINTS_SURF = "..\\database\\vision\\surf\\keypoints\\*"
FILE_PATH_LOAD_DESCRIPTORS_SURF = "..\\database\\vision\\surf\\descriptors\\*"
FILE_PATH_IMAGE_LOAD_SURF = "..\\database\\vision\\surf\\images\\*"

# calculates the keypoint, descriptors and saves it to the database files
def create_file_database(type_alg, image_path, img, kpt, des):
```

```python
    #get image name, without complete path
   img_filename, _ = os.path.splitext(image_path)
   file_basename = basename(img_filename)

   if(type_alg == 'sift'):
       file_kp = FILE_PATH_KEYPOINTS_SIFT + file_basename
       file_desc = FILE_PATH_DESCRIPTORS_SIFT + file_basename
       file_img = FILE_PATH_IMAGE_SIFT + file_basename

   elif(type_alg == 'surf'):
       file_kp = FILE_PATH_KEYPOINTS_SURF + file_basename
       file_desc = FILE_PATH_DESCRIPTORS_SURF + file_basename
       file_img = FILE_PATH_IMAGE_SURF + file_basename
   else:
       print("Unknown algorithm when creating database.", flush=True)
       return

   pickle_tmp = pickle_keypoints(kpt)

   with open(file_kp, 'wb') as fp:
       pickle.dump(pickle_tmp, fp)


   with open(file_desc, 'wb') as fp:
       pickle.dump(des, fp)


   with open(file_img, 'wb') as fp:
       pickle.dump(img, fp)

# load the keypoints, descriptors and images in the database
def load_fileImages_database(type_alg):

   print("Loading database...", flush=True)

   file_kp = ""
   file_desc = ""
   file_img = ""

   if(type_alg == 'sift'):
       file_kp = FILE_PATH_LOAD_KEYPOINTS_SIFT
       file_desc = FILE_PATH_LOAD_DESCRIPTORS_SIFT
       file_img = FILE_PATH_IMAGE_LOAD_SIFT

   elif(type_alg == 'surf'):
       file_kp = FILE_PATH_LOAD_KEYPOINTS_SURF
       file_desc = FILE_PATH_LOAD_DESCRIPTORS_SURF
       file_img = FILE_PATH_IMAGE_LOAD_SURF
   else:
       print("Unknown algorithm", flush=True)
       return

   file_list_keypoints = []
   for filename in glob.glob(file_kp):
       file_list_keypoints.append(filename)

   file_list_descriptors = []
   for filename in glob.glob(file_desc):
       file_list_descriptors.append(filename)

   file_list_image = []
   for filename in glob.glob(file_img):
       file_list_image.append(filename)



   #Create database if not exist
   if len(file_list_keypoints) < 1:
       print("no database for feature_points and descriptors", flush=True)
   elif len(file_list_image) < 1:
       print("no database for images cv", flush=True)
   elif(len(file_list_keypoints) != len(file_list_image)
       | len(file_list_keypoints) != len(file_list_descriptors)
       | len(file_list_image) != len(file_list_descriptors) ):

       print("database inconsistency", flush=True)
   else:

       all_images_cv = []
       for file in file_list_image:
           with open (file, 'rb') as fp:
               images_cv = pickle.load(fp)
               all_images_cv.append(images_cv)

       all_descriptors = []

       for file in file_list_descriptors:
           with open (file, 'rb') as fp:
               desc_tmp = pickle.load(fp)
               all_descriptors.append(desc_tmp)

       all_feature_points = []

       for file in file_list_keypoints:
           with open (file, 'rb') as fp:
               temp_kp = pickle.load(fp)

           feature_points = []

           for list_kp in temp_kp:
               temp_feature = unpickle_keypoints(list_kp)
               feature_points.append(temp_feature)

           all_feature_points.append(feature_points)

       print("LOAD COMPLETO", flush=True)
       return all_images_cv, all_feature_points, all_descriptors

# delete the saved images in the database
def deleteImageFromDatabase(image, name):

   try:
       #Remove image
       os.remove(DATABASE_PATH_IMAGES + image)

       #Remove layer
       os.remove(DATABASE_PATH_IMAGES_LAYERS + name + '_layer.png')

       #Remove SIFT
       os.remove(FILE_PATH_KEYPOINTS_SIFT + name)
       os.remove(FILE_PATH_DESCRIPTORS_SIFT + name)
```

```
        os.remove(FILE_PATH_IMAGE_SIFT + name)

        #Remove SURF
        os.remove(FILE_PATH_KEYPOINTS_SURF + name)
        os.remove(FILE_PATH_DESCRIPTORS_SURF + name)
        os.remove(FILE_PATH_IMAGE_SURF + name)

    except OSError:
        pass
```

```
import cv2
import numpy as np
from vision.database import create_file_database

descriptors = None

# Computes the mask select by the user
def applyMask(img, r, debug_bool):

    # Create the basic black image
    mask = np.zeros(img.shape, dtype = "uint8")

    # Draw a white, filled rectangle on the mask image
    x1 = r[0]
    y1 = r[1]
    x2 = x1 + r[2]
    y2 = y1 + r[3]
    mask_color = cv2.rectangle(mask, (x1, y1), (x2, y2), (255, 255, 255), -1)
    mask_color = cv2.cvtColor(mask_color,cv2.COLOR_BGR2GRAY)
    res = cv2.bitwise_and(img,img,mask = mask_color)

    if debug_bool:
        # Display constructed mask
        cv2.namedWindow("Mask", cv2.WINDOW_NORMAL)
        cv2.resizeWindow("Mask", 600, 400)
        cv2.imshow("Mask", mask)
        cv2.namedWindow("Res", cv2.WINDOW_NORMAL)
        cv2.resizeWindow("Res", 600, 400)
        cv2.imshow("Res", res)

        cv2.waitKey(0)

    return res

# Calculates the Keypoints and the descriptors of the selected mask, and store
them into the respective files
def select_region(image_path, debug_bool):
    global descriptors

    # Read image
    im = cv2.imread(image_path)

    # Select ROI
    cv2.namedWindow('keypoints', cv2.WINDOW_NORMAL)
    cv2.resizeWindow('keypoints', 600,600)
    r = cv2.selectROI('keypoints', im, True)

    res = applyMask(im, r, debug_bool)

    print('Calculating feature points for image %s' % image_path, flush=True)
    cv2.destroyAllWindows()
    gray = cv2.cvtColor(res, cv2.COLOR_BGR2GRAY)

    print('sift algorithm',flush=True)
    # Initiate SIFT detector
    sift = cv2.xfeatures2d.SIFT_create()
    # find the keypoints and descriptors with SIFT
    kp1, des1 = sift.detectAndCompute(gray,None)

    descriptors = des1
    removeKeyPoints(gray, kp1)
    des1 = descriptors

    if debug_bool:
        print('kp %s desc %s ' % (len(kp1),len(des1)) ,flush=True)
    create_file_database('sift', image_path, im, kp1,des1)

    if debug_bool:
        print('Hessian Threshold = 400',flush=True)

    surf = cv2.xfeatures2d.SURF_create(400)
    # Find keypoints and descriptors directly
    kp, des = surf.detectAndCompute(gray,None)

    descriptors = des
    removeKeyPoints(gray, kp)
    des = descriptors

    if debug_bool:
        print('kp %s desc %s ' % (len(kp),len(des)) ,flush=True)

    create_file_database('surf', image_path, im, kp,des)

# Calculates the Keypoints and the descriptors for the entire image
def keypoints_default(image_path, debug_bool):
    # Read image
    im = cv2.imread(image_path)

    if debug_bool:
            print('Calculating feature points for image %s' % image_path,
flush=True)

    gray = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)

    print('sift algorithm',flush=True)
    # Initiate SIFT detector
    sift = cv2.xfeatures2d.SIFT_create()
    # find the keypoints and descriptors with SIFT
    kp1, des1 = sift.detectAndCompute(gray,None)

    if debug_bool:
        print('kp %s desc %s ' % (len(kp1),len(des1)) ,flush=True)

    create_file_database('sift', image_path, im, kp1,des1)
    print('surf algorithm',flush=True)

    if debug_bool:
        print('Hessian Threshold = 400',flush=True)

    surf = cv2.xfeatures2d.SURF_create(400)
    # Find keypoints and descriptors directly
    kp, des = surf.detectAndCompute(gray,None)

    if debug_bool:
        print('kp %s desc %s ' % (len(kp),len(des)) ,flush=True)

    create_file_database('surf', image_path, im, kp,des)

#Window to remove keypoints with mouse click
def removeKeyPoints(gray, kp1):

    keycode = -1
    closed = 0.0
    _color = (0,0,255)

    while (keycode != 13) & (closed != -1.0): #Enter pressed
        img_kp =cv2.drawKeypoints(gray, kp1, None, color=_color, flags=2)
        cv2.imshow('remove_keypoints',img_kp)
        cv2.setMouseCallback("remove_keypoints", click_and_delete, param=[kp1])
        keycode = cv2.waitKey(100)
        closed = cv2.getWindowProperty('remove_keypoints', 0)
    cv2.destroyWindow('remove_keypoints')

#Checks if coordinates are inside the circle: (center_x, center_y) and radius
def inside_circle(x, y, center_x, center_y, radius):
    if ((((x - center_x)**2) + ((y - center_y)**2)) < radius**2):
        return True

    return False

#Check if mouse input over keypoint and deletes it
def click_and_delete(event, x, y, flags, param):

    global descriptors
    kp = param[0]

    if event == cv2.EVENT_LBUTTONDOWN:
        print(x, y, flush=True)

        for n in range(len(kp)):
            center_x = kp[n].pt[0]
            center_y = kp[n].pt[1]
            radius = kp[n].size
            if(inside_circle(x, y, center_x, center_y, radius)):
                del(kp[n])
                descriptors = np.delete(descriptors, n, 0)
                break
```

```
import os.path
import glob
from PIL import Image
from os.path import splitext, basename
import cv2
import numpy as np

DATABASE_PATH = '..\database\images\*'
LAYER_AR_PATH = '..\database\layers\\'

#Get the greater image number as index
def get_image_index():
    index = 1
    files = glob.glob(DATABASE_PATH)

    if(len(files) !=0):
        filename = os.path.splitext(files[-1])[0]
        index = basename(filename).replace('img', '')
        index = int(index) + 1

    return index

#Get the number of files in database images
```

```
def get_number_of_files():

   files = glob.glob(DATABASE_PATH)

   return len(files)

# Gets the database image name at index
def get_image_layerAR(index):

   image_list = []
   for filename in glob.glob(DATABASE_PATH):
       image_list.append(filename)

   filename = image_list[index]
   layer_filename, _ = os.path.splitext(filename)

   layer_basename = basename(layer_filename)
   layer_filename = LAYER_AR_PATH + layer_basename + '_layer.png'

   return layer_filename

# makes a mask using the two images
def blend_transparent(face_img, overlay_t_img):
   # Split out the transparency mask from the colour info
   overlay_img = overlay_t_img[:,:,:3] # Grab the BRG planes
   overlay_mask = overlay_t_img[:,:,3:]  # And the alpha plane

   # Again calculate the inverse mask
   background_mask = 255 - overlay_mask

   # Turn the masks into three channel, so we can use them as weights
   overlay_mask = cv2.cvtColor(overlay_mask, cv2.COLOR_GRAY2BGR)
   background_mask = cv2.cvtColor(background_mask, cv2.COLOR_GRAY2BGR)

   # Create a masked out face image, and masked out overlay
   # We convert the images to floating point in range 0.0 - 1.0
   face_part = (face_img * (1 / 255.0)) * (background_mask * (1 / 255.0))
   overlay_part = (overlay_img * (1 / 255.0)) * (overlay_mask * (1 / 255.0))
```

```
    # And finally just add them together, and rescale it back to an 8bit integer image
    return np.uint8(cv2.addWeighted(face_part, 255.0, overlay_part, 255.0, 0.0))

# convertes keypoint into a byte stream
def pickle_keypoints(keypoints):
   temp_array = []
   for point in keypoints:
       temp = (point.pt, point.size, point.angle, point.response, point.octave,
       point.class_id)
       temp_array.append(temp)
   return temp_array

# the byte stream is converted back into an object hierarchy.
def unpickle_keypoints(point):
     temp_feature = cv2.KeyPoint(x=point[0][0],y=point[0][1],_size=point[1], _angle=point[2], _response=point[3], _octave=point[4], _class_id=point[5])
    return temp_feature
```

GUI Code

```
from lib import *
from PIL import ImageTk, Image
import pickle
import tkinter.ttk as ttk
from tkinter import colorchooser
import tkinter.font as tkFont

from os.path import basename

from vision.choose import select_region, keypoints_default
from vision.ar_labeling import arAppCompute
from vision.utils import get_number_of_files, get_image_index
import vision.database as vdb
from gui.popWin import *
import gui.palette as palette
from gui.showDatabase import *


#Funciones intrinsecas de Paint
def isNumerable(x):
   if x.strip().isdigit():
       return True
   else:
       return False

#Constantes de carpetas
DATAFOLDER = "Data/"
DATASAVES = DATAFOLDER+"saves/"
DATADOCS = DATAFOLDER+"docs/"
DATALANG = DATAFOLDER+"langs/"
DATAICONS = DATAFOLDER+"icons/"
TEST_PATH = '../database/sample/'
DATABASE_PATH = '../database/images/'
DATABASE_LAYERS = '../database/layers/'

#Program Information
```

```
VERSION = 1.0,1.0
AUTOR = "\nInês Caldas\nJoel Carneiro"
PROGRAM_TITLE = "MonumentAR"

#Configuration file
CONFIGURATION_FILE = PROGRAM_TITLE+".ini"

#Default configuration
C_DATA                                =                                [[1048,
768],"#000000","#FFFFFF","#FFFFFF",[5,5,[1,1],0,"miter"],2,3,"EN"]

#Load Settings and Update C_DATA
try:
   conf_file = open(CONFIGURATION_FILE,"r")
   for i in conf_file:
       i = i.strip()
       c_command = i.split("=")
       if c_command[0].strip()=="PROGRAM_SIZE":
           c_after_command = str(c_command[1]).split(",")
                                        if    isNumerable(c_after_command[0]):
C_DATA[0][0]=int(c_after_command[0])
                                        if    isNumerable(c_after_command[1]):
C_DATA[0][1]=int(c_after_command[1])
                                 if     c_command[0].strip()=="DEFAULT_COLOR":
C_DATA[1]=str(c_command[1]).upper().strip()
                                if     c_command[0].strip()=="DEFAULT_ERASER":
C_DATA[2]=str(c_command[1]).upper().strip()
                             if     c_command[0].strip()=="DEFAULT_BACKGROUND":
C_DATA[3]=str(c_command[1]).upper().strip()
       if c_command[0].strip()=="DEFAULT_TOOL_STYLE":
                                                          c_after_command      =
(c_command[1].strip().replace("[","").replace("]","")).split(",")
                         if  isNumerable(c_after_command[0]):  C_DATA[4][0]   =
int(c_after_command[0])
                         if  isNumerable(c_after_command[1]):  C_DATA[4][1]   =
int(c_after_command[1])
                       if  isNumerable(c_after_command[2]):  C_DATA[4][2][0]   =
int(c_after_command[2])
                       if  isNumerable(c_after_command[3]):  C_DATA[4][2][1]   =
int(c_after_command[3])
```

```
                         if isNumerable(c_after_command[4]): C_DATA[4][3] =
int(c_after_command[4])
              if len(c_after_command[5])>0 and (c_after_command[5]=="miter" or
c_after_command[5]=="bevel" or c_after_command[5]=="round"):
               C_DATA[4][4] = c_after_command[5].replace("\"","").lower()
       if c_command[0].strip()=="DEFAULT_TOOL_WEIGHT":
           if isNumerable(c_command[1]): C_DATA[5]=int(c_command[1])
       if c_command[0].strip()=="DEFAULT_TOOL":
           if isNumerable(c_command[1]): C_DATA[6]=int(c_command[1])
       if c_command[0].strip()=="DEFAULT_LANGUAGE":
           C_DATA[7]=str(c_command[1]).strip().upper()
   conf_file.close()
except:
   lib("error","kernel",[1])
   lib("sonido","fatal")
   try:
       lib("conf_file",False,[CONFIGURATION_FILE,C_DATA])
       print ("IO/MESSAGE: New configuration file generated")
   except:
       print ("ERROR - 0,1: Cannot create configuration file")

#Default constants
DEFAULT_TITLE = "MonumentAR"
DEFAULT_EXTENSION = ".eps"

#Variables Configuration
PROGRAM_SIZE = C_DATA[0]
DEFAULT_COLOR = C_DATA[1]
DEFAULT_ERASER = C_DATA[2]
DEFAULT_BACKGROUND = C_DATA[3]
DEFAULT_TOOL_STYLE = C_DATA[4]
DEFAULT_TOOL_WEIGHT = C_DATA[5]
DEFAULT_TOOL = C_DATA[6]


LINE, OVAL, RECTANGLE, TEXT = list(range(4))
PENCIL, BRUSH = list(range(2))

#Class paint
class Paint:
   #Constructor
   def __init__(self):

       try:
           #Draw variables
           self.pos = [[0,0],[0,0]]
           self.activeFigure = None
           self._obj, self._objSave = None, None
           self.lastx, self.lasty = None, None
           self.vertices = 0
           self.pointable = []
           self.befpoint = [0,0]
           self.title = DEFAULT_TITLE
           self.activeTool = DEFAULT_TOOL
           self.activeColor = DEFAULT_COLOR
           self.backgroundColor = DEFAULT_COLOR
           self.toolWeight = DEFAULT_TOOL_WEIGHT
                                                          self.toolStyle     =
[DEFAULT_TOOL_STYLE[0],DEFAULT_TOOL_STYLE[1],DEFAULT_TOOL_STYLE[2],\

DEFAULT_TOOL_STYLE[3],DEFAULT_TOOL_STYLE[4]]
           self.draw = False
           self.mainArchive = ""
           self.imageBackgroundPath = ""
           self.layerName = ""
           #Elements draw on canvas
           self.stackElements = []
           self.stackElementsSave = []
           #Database images
           self.sizeDatabase = get_image_index()

           self.command = []
           #Window Creation
           self.main = Tk()
           #style = ttk.Style()
           #style.configure('TButton', background='black')
           #style.configure('TButton', foreground='green')
                #('winnative', 'clam', 'alt', 'default', 'classic', 'vista',
'xpnative')
```

```
           #style.theme_use("xpnative")
           #print(style.theme_names())
           self.main.focus_force()
                        self.main.geometry('%dx%d+%d+%d'  %  (PROGRAM_SIZE[0],
PROGRAM_SIZE[1], (self.main.winfo_screenwidth() - PROGRAM_SIZE[0])/2,\

(self.main.winfo_screenheight() - PROGRAM_SIZE[1])/2))
           self.main.title(PROGRAM_TITLE)
           self.main.iconbitmap(DATAICONS+"coloricon.ico")
           self.main.minsize(PROGRAM_SIZE[0], PROGRAM_SIZE[1])
           self.main.resizable(width=False, height=False)

           #Window events
           self.main.bind("<Control-Q>",self.exit)
           self.main.bind("<Control-q>",self.exit)
           self.main.bind("<Control-N>",self.newImage)
           self.main.bind("<Control-n>",self.newImage)
           self.main.bind("<Control-s>",self.saveImageLayer)
           self.main.bind("<Control-S>",self.saveImageLayer)
           self.main.bind("<Control-h>",self.help)
           self.main.bind("<Control-H>",self.help)



           #Menu
           menuBar = Menu(self.main)
           self.main.config(menu=menuBar)

           #File
           fileMenu = Menu(menuBar,tearoff=0)
           fileMenu.add_command(label="New    [Ctrl-N]",command=self.newImage)
                                              fileMenu.add_command(label="Save
[Ctrl-S]",command=self.saveImageLayer)
           fileMenu.add_separator()
           fileMenu.add_command(label="Exit    [Ctrl-Q]",command=self.exit)
           menuBar.add_cascade(label="File",menu=fileMenu)

           #Settings
           settingsMenu = Menu(menuBar,tearoff=0)
           colorMenu = Menu(settingsMenu,tearoff=0)
           settingsMenu.add_cascade(label="Change Color",menu=colorMenu)
```

```
                                            colorMenu.add_command(label="Color
1",command=lambda:self.colorChange("active"))
                                            colorMenu.add_command(label="Color
2",command=lambda:self.colorChange("background"))
                                          settingsMenu.add_command(label="Tool
Weight",command=self.toolWeightChange)
           menuBar.add_cascade(label="Settings",menu=settingsMenu)

           #Insert
           insertMenu = Menu(menuBar,tearoff=0)
                   #TODO:  insertMenu.add_command(label="Arc",command=  lambda:
self.createFigure("arc"))
                       insertMenu.add_command(label="Square",command=  lambda:
self.createFigure("square"))
                         insertMenu.add_command(label="Oval",command=  lambda:
self.createFigure("oval"))
                         insertMenu.add_command(label="Line",command=  lambda:
self.createFigure("line"))
                         insertMenu.add_command(label="Text",command=  lambda:
self.createFigure("text"))
                           insertMenu.add_command(label="Icon",command=lambda:
self.insertIcons())

           menuBar.add_cascade(label="Insert",menu=insertMenu)

           #Tools
           toolsMenu = Menu(menuBar,tearoff=0)
           tMenu = Menu(toolsMenu,tearoff=0)

tMenu.add_command(label="Pencil",command=lambda:self.tools("pencil"))
           tMenu.add_command(label="Brush",command=lambda:self.tools("brush"))
           menuBar.add_cascade(label="Tools",menu=tMenu)


           #Help
           Help = Menu(menuBar,tearoff=0)
           Help.add_command(label="About",command=self.about)
           Help.add_command(label="Help    [Ctrl-h]",command=self.help)
           Help.add_command(label="Changelog",command=self.changelog)
           Help.add_command(label="License",command=self.license)
```

```
        menuBar.add_cascade(label="Help",menu=Help)

        #Draw Frame
        ParentFrame = Frame(self.main, background=palette.BACKGROUND_WINDOW)
        ParentFrame.grid()

        #Draw Canvas
        windowFrame = Frame(ParentFrame, background=palette.BACKGROUND_WINDOW)
        windowFrame.grid_rowconfigure(0, weight=1)
        windowFrame.grid_columnconfigure(0, weight=1)
        windowFrame.grid(row=0, column=0, sticky="nsew")
        windowFrame2 = Frame(ParentFrame, background=palette.BACKGROUND_WINDOW)
        windowFrame2.grid_rowconfigure(0, weight=1)
        windowFrame2.grid_columnconfigure(0, weight=1)
        windowFrame2.grid(row=0, column=0, sticky="nsew")

        windowFrame2.lower()

        self.screen = Canvas(windowFrame,width=PROGRAM_SIZE[0]*0.7815, height=PROGRAM_SIZE[1],bg=palette.CANVAS_COLOR)
        self.screenSave = Canvas(windowFrame2,width=PROGRAM_SIZE[0]*0.7815, height=PROGRAM_SIZE[1],bg=DEFAULT_BACKGROUND, relief="sunken")
        self.screen.grid()
        self.screenSave.grid()

        #Buttons
        Buttonframe = Frame(ParentFrame,border=5, background=palette.BACKGROUND_WINDOW)
        Buttonframe.grid(row=0, column=1, sticky="NW")
        label = Label(Buttonframe,text="Tools",border=10, background=palette.BACKGROUND_WINDOW, fg=palette.LIGHT_GRAY)
        label.config(font=("Courier", 18, 'bold'))
        label.pack()

        #Tools

        ToolsFrame = Frame(Buttonframe)
```

```
           ToolsFrame.pack()

                                                                             b_undo       =
ttk.Button(ToolsFrame,text="Undo",width=20,command=self.undoElement,
style="TButton")
           image_undo = Image.open(DATAICONS + "eraser.png")
           image_undo = image_undo.resize((32,32), Image.ANTIALIAS)
           image_undo = ImageTk.PhotoImage(image_undo)
           b_undo.config(image=image_undo)
           b_undo.pack(side=LEFT)

                                                                           b_pencil       =
ttk.Button(ToolsFrame,text="Pencil",width=20,command=lambda:self.tools("pencil
"), style="TButton")
           image_pencil = Image.open(DATAICONS + "pencil.png")
           image_pencil = image_pencil.resize((32,32), Image.ANTIALIAS)
           image_pencil = ImageTk.PhotoImage(image_pencil)
           b_pencil.config(image=image_pencil)
           b_pencil.pack(side=LEFT)

                                                                            b_brush       =
ttk.Button(ToolsFrame,text="Brush",width=20,command=lambda:self.tools("brush")
, style="TButton")
           image_brush = Image.open(DATAICONS + "brush.png")
           image_brush = image_brush.resize((32,32), Image.ANTIALIAS)
           image_brush = ImageTk.PhotoImage(image_brush)
           b_brush.config(image=image_brush)
           b_brush.pack(side=LEFT)

                 b_text  =  ttk.Button(ToolsFrame,text="Text",width=20,command=
lambda: self.createFigure("text"), style="TButton")
           image_text = Image.open(DATAICONS + "text.png")
           image_text = image_text.resize((32,32), Image.ANTIALIAS)
           image_text = ImageTk.PhotoImage(image_text)
           b_text.config(image=image_text)
           b_text.pack(side=LEFT)

           #Insert Figures
                                           FiguresInsert    =    Frame(Buttonframe,
background=palette.BACKGROUND_WINDOW)
```

```
        FiguresInsert.pack()

        b_line = ttk.Button(FiguresInsert,text="Insert Line",width=20,command=lambda:self.createFigure('line'), style="TButton")
        image_line = Image.open(DATAICONS + "line.png")
        image_line = image_line.resize((32,32), Image.ANTIALIAS)
        image_line = ImageTk.PhotoImage(image_line)
        b_line.config(image=image_line)
        b_line.pack(side=LEFT)

        b_square = ttk.Button(FiguresInsert,text="Insert Square",width=20,command=lambda:self.createFigure('square'), style="TButton")
        image_square = Image.open(DATAICONS + "square.png")
        image_square = image_square.resize((32,32), Image.ANTIALIAS)
        image_square = ImageTk.PhotoImage(image_square)
        b_square.config(image=image_square)
        b_square.pack(side=LEFT)

        b_oval = ttk.Button(FiguresInsert,text="Insert Oval",width=20,command=lambda:self.createFigure('oval'), style="TButton")
        image_oval = Image.open(DATAICONS + "circle.png")
        image_oval = image_oval.resize((32,32), Image.ANTIALIAS)
        image_oval = ImageTk.PhotoImage(image_oval)
        b_oval.config(image=image_oval)
        b_oval.pack(side=LEFT)

        b_icon = ttk.Button(FiguresInsert,text="Insert Icons",width=20,command=self.insertIcons, style="Wild.TButton")
        image_icon = Image.open(DATAICONS + "icon.png")
        image_icon = image_icon.resize((32,32), Image.ANTIALIAS)
        image_icon = ImageTk.PhotoImage(image_icon)
        b_icon.config(image=image_icon)
        b_icon.pack(side=LEFT)

        #Tools info
        label = Label(Buttonframe,text="Settings",border=10, background=palette.BACKGROUND_WINDOW, fg=palette.LIGHT_GRAY)
        label.config(font=("Courier", 18, 'bold'))
        label.pack()
```

```
                                        WeightPencil    =    Frame(Buttonframe,
background=palette.BACKGROUND_WINDOW, width=32, height=32)
           WeightPencil.pack()
                   self.infoWeightPencil  =  Label(WeightPencil,relief='groove'
,text=str(self.toolWeight),border=3,font=10,width=3,
background=palette.CANVAS_COLOR, fg=palette.LIGHT_GRAY)
           self.infoWeightPencil.config(height=2, width=3)
           self.infoWeightPencil.pack(side=LEFT)


                                                                b_weight      =
ttk.Button(WeightPencil,text="Weight",command=self.toolWeightChange,width=2)
           image_weight = Image.open(DATAICONS + "weight.png")
           image_weight = image_weight.resize((32,32), Image.ANTIALIAS)
           image_weight = ImageTk.PhotoImage(image_weight)
           b_weight.config(image=image_weight)
           b_weight.pack()


           #Color Information
                                         activeColor    =    Frame(Buttonframe,
background=palette.BACKGROUND_WINDOW)
           activeColor.pack()
                                                   self.infoactivedcolor      =
Canvas(activeColor,width=32,height=32,bg=self.activeColor)
           self.infoactivedcolor.pack(side=LEFT)

                               b_color   =   ttk.Button(activeColor,text="Color
1",command=lambda:self.colorChange("active"),width=32)
           image_color = Image.open(DATAICONS + "paint.png")
           image_color = image_color.resize((32,32), Image.ANTIALIAS)
           image_color = ImageTk.PhotoImage(image_color)
           b_color.config(image=image_color)
           b_color.pack()

                                         activeColor    =    Frame(Buttonframe,
background=palette.BACKGROUND_WINDOW)
           activeColor.pack()
                                          self.infoactivedbackgroundcolor     =
Canvas(activeColor,width=32,height=32,bg=self.backgroundColor)
```

```
            self.infoactivedbackgroundcolor.pack(side=LEFT)

                    b_colorBucket = ttk.Button(activeColor,text="Color
2",command=lambda:self.colorChange("background"),width=10)
            image_color_bucket = Image.open(DATAICONS + "paint2.png")
                 image_color_bucket = image_color_bucket.resize((32,32),
Image.ANTIALIAS)
            image_color_bucket = ImageTk.PhotoImage(image_color_bucket)
            b_colorBucket.config(image=image_color_bucket)
            b_colorBucket.pack()

                            activeColor = Frame(Buttonframe,
background=palette.BACKGROUND_WINDOW)
            activeColor.pack()




            #Vision Buttons
                 label = Label(Buttonframe,text="Vision",border=10, bg =
palette.BACKGROUND_WINDOW, fg=palette.LIGHT_GRAY)
            label.config(font=("Courier", 18, 'bold'))
            label.pack()
                              ttk.Button(Buttonframe,text="Add
image",width=20,command=self.addImageDatabase, style="TButton").pack()
                              ttk.Button(Buttonframe,text="Key
Points",width=20,command=self.computeKeyPoints).pack()

ttk.Button(Buttonframe,text="SIFT",width=20,command=lambda:self.arApp('sift'))
.pack()

ttk.Button(Buttonframe,text="SURF",width=20,command=lambda:self.arApp('surf'))
.pack()

ttk.Button(Buttonframe,text="Database",width=20,command=self.seeDatabase).pack
()

                   label = Label(Buttonframe,text="",border=10, bg =
palette.BACKGROUND_WINDOW, fg=palette.LIGHT_GRAY)
            label.config(font=("Courier", 5, 'bold'))
            label.pack()
```

```
           self.ransac_value = Scale(Buttonframe, from_=0, to=50,
label='Ransac Threshold', width=20, resolution=0.1, orient=HORIZONTAL,
bg=palette.BACKGROUND_WINDOW, fg=palette.LIGHT_GRAY)
           self.ransac_value.pack(fill=BOTH)
           self.ransac_value.set(0.6)

           #Info for user
           Label(Buttonframe,height=1, background=palette.CANVAS_COLOR).pack()
           self.messageUser =
Label(Buttonframe,text="",relief=GROOVE,width=30,height=5,justify=CENTER,wrapl
ength=125, background=palette.CANVAS_COLOR)
           self.messageUser.config(fg=palette.LIGHT_GRAY)
           self.messageUser.pack()

           #Debug
           self.debug = BooleanVar()
           c = Checkbutton(Buttonframe, text="Debug",
selectcolor=palette.CANVAS_COLOR, variable=self.debug, bg =
palette.BACKGROUND_WINDOW, fg=palette.LIGHT_GRAY,
command=self.changeDebugMode)
           c.pack(side=RIGHT)
           c.var = self.debug

           # add bindings for clicking, dragging and releasing over
           # any object with the "token" tag

           # this data is used to keep track of an
           # item being dragged
           self._drag_data = {"x": 0, "y": 0, "item": None, "itemSave": None}
           self.screen.tag_bind("token", "<ButtonPress-3>",
self.on_token_press)
           self.screen.tag_bind("token", "<ButtonRelease-3>",
self.on_token_release)
           self.screen.tag_bind("token", "<B3-Motion>", self.on_token_motion)


           #Init functions indev
           self.tools(self.activeTool)
           self.screen.bind("<ButtonRelease-1>",self.posPointer)
```

```
            #Window is created
            self.main.mainloop(0)

        except ValueError:
            print(ValueError)
            #lib("error","kernel",[2])
            #lib("error","kernel",[3])

    #Undo Elements
    def undoElement(self):
        if(len(self.stackElements) > 0):
            element = self.stackElements.pop()
            self.screen.delete(element)
            self.draw = True
        if(len(self.stackElementsSave) > 0):
            element = self.stackElementsSave.pop()
            self.screenSave.delete(element)
            self.draw = True

    #Free draw
    def freeDraw(self,event):

        if self.activeTool==PENCIL or self.activeTool==BRUSH:
            colorpaint = self.activeColor

        if self.toolWeight==1:
            if self.befpoint==[0,0]:
                self.befpoint = [event.x,event.y]

                                                                element      =
self.screen.create_line(event.x,event.y,self.befpoint[0]+self.toolStyle[0]-DEF
AULT_TOOL_STYLE[0],self.befpoint[1]+\
                                      self.toolStyle[1]-DEFAULT_TOOL_STYLE[1],
dash=self.toolStyle[2],\

width=self.toolWeight,fill=colorpaint,smooth=self.toolStyle[3])
                                                               elementS      =
self.screenSave.create_line(event.x,event.y,self.befpoint[0]+self.toolStyle[0]
-DEFAULT_TOOL_STYLE[0],self.befpoint[1]+\
                                      self.toolStyle[1]-DEFAULT_TOOL_STYLE[1],
dash=self.toolStyle[2],\

width=self.toolWeight,fill=colorpaint,smooth=self.toolStyle[3])

           self.stackElements.append(element)
           self.stackElementsSave.append(elementS)

           self.befpoint = [event.x,event.y]

       else:
           if self.activeTool==BRUSH:
                                                                 elementS     =
self.screenSave.create_rectangle(event.x,event.y,event.x+self.toolStyle[0],eve
nt.y+\
                                 self.toolStyle[1],dash=self.toolStyle[2],\
                                 width=self.toolWeight,fill=colorpaint,outline
= colorpaint)
                                                                  element     =
self.screen.create_rectangle(event.x,event.y,event.x+self.toolStyle[0],event.y
+\
                                 self.toolStyle[1],dash=self.toolStyle[2],\
                                 width=self.toolWeight,fill=colorpaint,outline
= colorpaint)

               self.stackElements.append(element)
               self.stackElementsSave.append(elementS)

           else:
                                                                  element     =
self.screen.create_line(event.x,event.y,event.x+self.toolStyle[0],event.y+\
                                 self.toolStyle[1],dash=self.toolStyle[2],\

width=self.toolWeight,fill=colorpaint,smooth=self.toolStyle[3])
                                                                 elementS     =
self.screenSave.create_line(event.x,event.y,event.x+self.toolStyle[0],event.y+
\
                                 self.toolStyle[1],dash=self.toolStyle[2],\

width=self.toolWeight,fill=colorpaint,smooth=self.toolStyle[3])
```

```
                self.stackElements.append(element)
                self.stackElementsSave.append(elementS)

        self.draw = True

    #New image database menu bar
    def newImage(self,i="null"):
        if self.draw:
            resp = popWin("Save",DATAICONS+"alert.ico","save",(250,80))
            resp.root.mainloop(1)
            if resp.value!=0:
                if resp.value: self.saveImageLayer()
        self.messageUser.config(text="")
        self.addImageDatabase()
        self.draw = False

    #Save imagen
    def saveImageLayer(self,i="null"):#TODO
        if self.draw:
            self.screen.update()
            self.screenSave.update()
            txt = popWin("Save",DATAICONS+"save.ico","save",(250,110))
            txt.root.mainloop(1)
            print(txt.value, flush=True)

            if txt.value:
                filename = DATASAVES+'tmp'+DEFAULT_EXTENSION
                print('as', self.screenSave.size, flush=True)
                   self.screenSave.postscript(file=filename, colormode='color', 
height = 770, pagewidth=819)

                img = Image.open(filename)
                print('size ', img.size, flush=True)
                self.saveLayer(img)
                img.save(DATASAVES+'tmp.png', 'png')
                self.draw = False

    #Save layer AR
    def saveLayer(self, img):
        img = img.convert("RGBA")
        datas = img.getdata()
        newData = []
        for item in datas:
            if item[0] == 255 and item[1] == 255 and item[2] == 255:
                newData.append((255, 255, 255, 0))
            else:
                newData.append(item)
        img.putdata(newData)
        layerPath = DATABASE_LAYERS + self.layerName + '_layer.png'
        print('n', layerPath, flush=True)
        img.save(layerPath, "PNG")#converted Image name

    #exit the program
    def exit(self,i="null"):
        if self.draw:
            resp = popWin("Save",DATAICONS+"alert.ico","saveIt",(250,80))
            resp.root.mainloop(1)
            if resp.value!=0:
                if resp.value: self.saveImageLayer()
        self.main.destroy()

    #Save Color Tools    - active, eraser
    def colorChange(self,tools):
        color = askcolor()
        color = color[1]
        if((color == '#ffffff' ) | (color == '#FFFFFF')):
            color = '#fefefe'

        if color!=0:
            if tools=="active":
                self.activeColor = color
                self.infoactivedcolor.config(bg=self.activeColor)
            if tools=="eraser":
                self.eraserColor = color
                self.infoactivedcoloreraser.config(bg=self.eraserColor)
            if tools=="background":
                self.backgroundColor = color
                self.infoactivedbackgroundcolor.config(bg=self.backgroundColor)
    #Change weight of tools
    def toolWeightChange(self):
        a = popWin("Tools Weight",DATAICONS+"grosor.ico","weight",(260,450))
        a.root.mainloop(1)
        if a.value!=0:
            self.toolWeight = a.value
            self.infoWeightPencil.config(text=str(a.value))



    #Insert Icons
    def insertIcons(self,E=False):
        a = iconWin(self.main, "Insert
icons",DATAICONS+"shaperound.ico","icons", (230,460))
        a.root.mainloop(0)
        self._create_icon(a.value)

    #Create Text
    def createText(self,event):
        txt = popWin("Write text",DATAICONS+"text.ico","inserttext",(250,110))
        txt.root.mainloop(1)
        self.messageUser.config(text="")
        _font = tkFont.Font(size = self.toolWeight + 10, weight='bold')
         _obj = self.screen.create_text(event.x,event.y, text=txt.value,font =
_font, fill=self.activeColor,activefill='red', justify=tk.CENTER,
tags='token')
        _objSave = self.screenSave.create_text(event.x,event.y,
text=txt.value,font = _font, fill=self.activeColor,activefill='red',
justify=tk.CENTER, tags='token')

        self.stackElements.append(_obj)
        self.stackElementsSave.append(_objSave)

        self.screen.bind("<ButtonPress-1>",self.breakpoint)
        self.draw = True

    #Create figures
    def createFigure(self,figura):
        if figura=="square":
            self.screen.bind("<ButtonPress-1>", self.update_xy)
            self.screen.bind("<B1-Motion>", self.drawFigure)
            self.activeFigure = RECTANGLE
            self.messageUser.config(text="Drag to create rectangle")
        if figura=="oval":
            self.screen.bind("<ButtonPress-1>",self.update_xy)
            self.screen.bind("<B1-Motion>",self.drawFigure)
            self.activeFigure = OVAL
            self.messageUser.config(text="Drag to create oval")
        if figura=="line":
            self.screen.bind("<ButtonPress-1>", self.update_xy)
            self.screen.bind("<B1-Motion>", self.drawFigure)
            self.messageUser.config(text="Drag to create line")
            self.activeFigure = LINE
        if figura=="text":
            self.activeFigure = TEXT
            self.screen.bind("<ButtonPress-1>",self.createText)
            self.messageUser.config(text="Click where to put text")

    #Change tools - eraser, pencil, brush
    def tools(self,herr):
        if herr=="pencil" or herr==1: self.activeTool=PENCIL
        if herr=="brush" or herr==3: self.activeTool=BRUSH
        self.screen.bind("<B1-Motion>",self.freeDraw)

    #Load window with help
    def help(self,i="null"):
                                                                a          =
popWin("help",DATAICONS+"help.ico","help",(600,400),[PROGRAM_TITLE,DATADOCS+"H
ELP.TXT"])
        a.root.mainloop(0)

    #Load About
    def about(self,i="null"):
                                                    a       =       popWin("About
"+PROGRAM_TITLE,DATAICONS+"coloricon.ico","about",(220,120),[AUTOR,VERSION[0]]
)
        a.root.mainloop(0)

    #Posicionar puntero
    def posPointer(self,event):
        self.befpoint=[0,0]
```

```
   #Lista de cambios del programa
   def changelog(self):
                                                                   a          =
popWin("Changelog",DATAICONS+"changelog.ico","changelog",(600,400),[PROGRAM_TI
TLE,DATADOCS+"CHANGELOG.TXT"])
       a.root.mainloop(0)

   #License of the program
   def license(self):
                                        a      =      popWin("Licencia     GNU
[English]",DATAICONS+"gnu.ico","license",(600,400),[PROGRAM_TITLE,DATADOCS+"GN
U.TXT"])
       a.root.mainloop(0)

   #Exit function
   def breakpoint(self,breakeable):
       return



   #TODO
   def drawFigure(self, event):
       if self.activeFigure is None or self._obj is None:
           return
       x, y = self.lastx, self.lasty
       if self.activeFigure in (LINE, RECTANGLE, OVAL):
           self.screen.coords(self._obj, (x, y, event.x, event.y))
           self.screenSave.coords(self._objSave, (x, y, event.x, event.y))

   def update_xy(self, event):
       if self.activeFigure is None:
           return
       x, y = event.x, event.y

       if self.activeFigure == LINE:
                        self._obj  =  self.screen.create_line((x,  y,  x,  y),
fill=self.activeColor,width=self.toolWeight, tags='token')
                  self._objSave  =  self.screenSave.create_line((x,  y,  x,  y),
fill=self.activeColor,width=self.toolWeight, tags='token')
```

```
        elif self.activeFigure == RECTANGLE:
            self._obj = self.screen.create_rectangle((x, y, x, y),
fill=self.backgroundColor,outline=self.activeColor, tags='token')
            self._objSave = self.screenSave.create_rectangle((x, y, x, y),
fill=self.backgroundColor,outline=self.activeColor, tags='token')

        elif self.activeFigure == OVAL:
            self._obj = self.screen.create_oval((x, y, x, y),
fill=self.backgroundColor,outline=self.activeColor, tags='token')
            self._objSave = self.screenSave.create_oval((x, y, x, y),
fill=self.backgroundColor,outline=self.activeColor, tags='token')

        elif self.activeFigure == TEXT:
            self._obj = self.screen.create_text(x, y,text='a',font="Arial",
tags='token')
            self._objSave = self.screen.create_text(x, y,text='a',font="Arial",
tags='token')

        element = self._obj
        elementS = self._objSave
        self.stackElements.append(element)
        self.stackElementsSave.append(elementS)
        self.draw = True
        self.lastx, self.lasty = x, y

    #Token Drag
    def _create_icon(self, filepath):
        if(filepath != None):
            print('aqui',filepath, flush=True)
            '''Create a icon at the given coordinate in the given color'''
            # load the .gif image file
            images = Image.open(filepath)
            images = images.resize((64,64), Image.ANTIALIAS)
            images = ImageTk.PhotoImage(images)
            im = self.screen.create_image(PROGRAM_SIZE[0]*0.7815/2,
PROGRAM_SIZE[1]/2, image=images, anchor=CENTER,tags="token", state=NORMAL)
            imSave = self.screenSave.create_image(PROGRAM_SIZE[0]*0.7815/2,
PROGRAM_SIZE[1]/2, image=images, anchor=CENTER,tags="token", state=NORMAL)
            self.stackElements.append(im)
            self.stackElementsSave.append(imSave)
            mainloop()
            self.draw = True

    def on_token_press(self, event):
        '''Begining drag of an object'''
        # record the item and its location
        self._drag_data["item"] = self.screen.find_closest(event.x, event.y)[0]
           self._drag_data["itemSave"] = self.screenSave.find_closest(event.x,
 event.y)[0]
        self._drag_data["x"] = event.x
        self._drag_data["y"] = event.y

    def on_token_release(self, event):
        '''End drag of an object'''
        # reset the drag information
        self._drag_data["item"] = None
        self._drag_data["itemSave"] = None
        self._drag_data["x"] = 0
        self._drag_data["y"] = 0

    def on_token_motion(self, event):
        if(self._drag_data["item"] != None):
            '''Handle dragging of an object'''
            # compute how much the mouse has moved
            delta_x = event.x - self._drag_data["x"]
            delta_y = event.y - self._drag_data["y"]
            # move the object the appropriate amount
            self.screen.move(self._drag_data["item"], delta_x, delta_y)
            self.screenSave.move(self._drag_data["itemSave"], delta_x, delta_y)
            # record the new position
            self._drag_data["x"] = event.x
            self._drag_data["y"] = event.y
            self.draw = True

    ###########################
    #Vision

    def addImageDatabase(self):
        self.messageUser.config(text="Enter the location of your image.")
        filepath = askopenfilename(title="Open",initialdir="./",defaultextension=".jpg",filetypes = (("jpeg files","*.jpg"),("all files","*.*")))
        self.messageUser.config(text="")

        if filepath!="": #TODO see if image file
            self.mainArchive=filepath

            filename, file_extension = os.path.splitext(filepath)

            image = Image.open(filepath)
            image = image.resize((int(PROGRAM_SIZE[0]*0.7815), PROGRAM_SIZE[1]), Image.ANTIALIAS)

            namefile = ""
            if(self.sizeDatabase<10):
                namefile = 'img0' + str(self.sizeDatabase)
            else:
                namefile = 'img' + str(self.sizeDatabase)

            name = DATABASE_PATH + namefile + file_extension
            image.save(name)
            self.imageBackgroundPath = name
            self.layerName = namefile
            image = ImageTk.PhotoImage(image)

            self.imageBackground = self.screen.create_image(0, 0, image = image, anchor = NW, tags='image')
            self.screenSave.delete(ALL)
            self.sizeDatabase = get_image_index()

            #Create empty layer
            self.screen.update()
            self.screenSave.update()

            filename = DATASAVES+'temp'+DEFAULT_EXTENSION
            print('as', self.screenSave.size, flush=True)
            self.screenSave.postscript(file=filename, colormode='color', height = 770, pagewidth=819)
            img = Image.open(filename)
            print(img.size, flush=True)
            self.saveLayer(img)
            img.save(DATASAVES+'temp' + '.png', 'png')

            #Calculates default keypoints and descriptors
            keypoints_default(self.imageBackgroundPath, self.debug.get())

            self.main.mainloop()

    def seeDatabase(self):
        num_files = get_number_of_files()

        if(num_files == 0):
            self.messageUser.config(text="Database empty.")

        else:
            database = showDatabase(self.main, DATABASE_PATH, DATABASE_LAYERS)
            database.root.mainloop(0)
            if(database.value != None):
                if(database.value[0] == 'edit'):
                    self.editLayer(database.value[1])
                elif(database.value[0] == 'delete'):
                    self.deleteImageDatabase(database.value[1])

    def editLayer(self, filepath):
        filename, file_extension = os.path.splitext(filepath)
        index = basename(filename).replace('img', '')

        image = Image.open(filepath)
          image = image.resize((int(PROGRAM_SIZE[0]*0.7815), PROGRAM_SIZE[1]),
 Image.ANTIALIAS)
        self.imageBackgroundPath = filepath
        self.layerName = 'img' + str(index)
        image = ImageTk.PhotoImage(image)

        self.screen.delete(ALL)
        self.screenSave.delete(ALL)
        self.stackElements = []
        self.stackElementsSave = []
        self.imageBackground = self.screen.create_image(0, 0, image = image, anchor = NW, tags='image')

        self.sizeDatabase = get_image_index()

        #Create empty layer
        self.screen.update()
        self.screenSave.update()

        self.main.mainloop(0)

    def deleteImageDatabase(self, filepath):
        filename, file_extension = os.path.splitext(filepath)
        index = basename(filename).replace('img', '')
        filepath = filepath.replace('\\','/')
        base_file = basename(filepath)
        print('delete a', filepath, self.imageBackgroundPath, flush=True)
        name = 'img' + str(index)

        if(filepath == self.imageBackgroundPath):
            print('delete curr', index, flush=True)
            self.cleanCanvas()

        vdb.deleteImageFromDatabase(base_file, name)
     def cleanCanvas(self):
        self.imageBackground = None
        self.imageBackgroundPath = ""
        self.layerName = ""
        self.sizeDatabase = get_image_index()

        self.screen.delete(ALL)
        self.screenSave.delete(ALL)
        self.stackElements = []
        self.stackElementsSave = []

        #Create empty layer
        self.screen.update()
        self.screenSave.update()
        self.main.mainloop(0)

    def computeKeyPoints(self):
        print("Aqui", flush=True)
        print("imageBackgroundPath", self.imageBackgroundPath)
        if(self.imageBackgroundPath==""):
            self.messageUser.config(text="Select image first.")
            print("Select image first.")
        else:
            select_region(self.imageBackgroundPath, self.debug.get())

    def arApp(self, algorithm):
        num_files = get_number_of_files()

        if(num_files == 0):
            self.messageUser.config(text="Database empty.")

        else:
            filepath =
askopenfilename(title="Open",initialdir=TEST_PATH,defaultextension=".jpg",file
types = (("jpeg files","*.jpg"),("all files","*.*")))
            self.messageUser.config(text="")
            #if filepath!="" and
(filepath[len(filepath)-4:len(filepath)]==".jpg" or
filepath[len(filepath)-4:len(filepath)]==".gif"):
            if filepath!="":
                print(filepath, flush=True)
                    arAppCompute(filepath, algorithm, self.ransac_value.get(),
self.debug.get())

    def changeDebugMode(self):
        if(self.debug.get()):
            print('Debug mode activated.', flush=True)
        else:
            print('Debug mode deactivated.', flush=True)


#Run class Paint
Paint()
```

```python
import glob
import os
import sys
import time

from PIL import Image, ImageTk

import tkinter as tk
import tkinter.ttk as ttk


def images(path):
    im = []

    for path in sys.path:
        im.extend(images_for(path))

    return sorted(im)


def images_for(path):
    if os.path.isfile(path):
        return [path]
    i = []
    for match in glob.glob("%s/*" % path):
        if match.lower()[-4:] in ('.jpg', '.png', '.gif'):
            i.append(match)
    return i

class showDatabase():
    def __init__(self, master, path, layerPath):
        self.master = master
        self.root = tk.Toplevel(master)
        self.value = None
        self.root.pack_propagate(False)
        self.root.config(bg="black", width=500, height=500)
        self._fullscreen = True
        self._images = images_for(path)
        self._imagesLayer = images_for(layerPath)
        self._image_pos = -1
        self.root.bind("<Return>", self.return_handler)
        self.root.bind("<space>", self.space_handler)
        self.root.bind("<Escape>", self.esc_handler)
        self.root.bind("<Left>", self.show_previous_image)
        self.root.bind("<Right>", self.show_next_image)
        self.root.bind("q", self.esc_handler)
        self.root.bind("f", self.f_handler)
        self.root.after(100, self.show_next_image)

        self.root.rowconfigure(3, minsize=500)
        self.root.columnconfigure(1, minsize=500)

        self.label = tk.Label(self.root, image=None)
        self.label.configure(borderwidth=0)
        self.label.grid(row=0, column=0, rowspan=2)
        tk.Button(self.root, text="New layer",
command=lambda:self.editLayer()).grid(row=2, column=0, sticky=tk.SE, padx=10)
        tk.Button(self.root, text="Delete",
command=lambda:self.deleteImage()).grid(row=2, column=0, sticky=tk.SE,
pady=30, padx=10)

        self.set_timer()

    slide_show_time = 4
    last_view_time = 0
    paused = False
    image = None

    def editLayer(self):
        self.value = ('edit', self._images[self._image_pos])
        self.root.quit()
        self.root.destroy()

    def deleteImage(self):
        self.value = ('delete', self._images[self._image_pos])
        self.root.quit()
        self.root.destroy()

    def f_handler(self, e):
        self._fullscreen = not self._fullscreen
        if self._fullscreen:
            self.root.attributes('-fullscreen', True)
        else:
            self.root.attributes('-fullscreen', False)
            self.root.attributes("-zoomed", True)

    def esc_handler(self, e):
        self.root.destroy()

    def return_handler(self, e):
        self.show_next_image()

    def space_handler(self, _):
        self.paused = not self.paused

    def set_timer(self):
        self.root.after(300, self.update_clock)

    def update_clock(self):
        if time.time() - self.last_view_time > self.slide_show_time \
           and not self.paused:
            self.show_next_image()
        self.set_timer()
        self.check_image_size()

    def show_next_image(self, e=None):
        fname, fnameLayer = self.next_image()
        if not fname:
            return
        self.show_image(fname, fnameLayer)

    def show_previous_image(self, e=None):
        fname, fnameLayer = self.previous_image()
        if not fname:
            return
        self.show_image(fname, fnameLayer)

    def show_image(self, fname, fnameLayer):
        self.original_image = Image.open(fname)
        self.original_image_layer = Image.open(fnameLayer)
        self.image = None
        self.fit_to_box()
        self.last_view_time = time.time()

    def check_image_size(self):
        if not self.image:
            return
        self.fit_to_box()

    def fit_to_box(self):
        if self.image:
            if self.image.size[0] == self.box_width: return
            if self.image.size[1] == self.box_height: return

        width, height = self.original_image.size
        new_size = scaled_size(width, height, self.box_width, self.box_height)
        self.image = self.original_image.resize(new_size, Image.ANTIALIAS)

        self.label.place(x=self.box_width/2, y=self.box_height/2, anchor=tk.CENTER)

        resized_image = self.original_image_layer.resize(new_size, Image.ANTIALIAS)

        self.image.paste(resized_image, (0,0), resized_image)

        tkimage = ImageTk.PhotoImage(self.image)
        self.label.configure(image=tkimage)
        self.label.image = tkimage


    @property
    def box_width(self):
        return self.root.winfo_width()

    @property
    def box_height(self):
        return self.root.winfo_height()
    def next_image(self):
        if not self._images:
            return None
        self._image_pos += 1
        self._image_pos %= len(self._images)
                                     return     self._images[self._image_pos],
self._imagesLayer[self._image_pos]

    def previous_image(self):
        if not self._images:
            return None
        self._image_pos -= 1
                                     return     self._images[self._image_pos],
self._imagesLayer[self._image_pos]

def scaled_size(width, height, box_width, box_height):
    source_ratio = width / float(height)
    box_ratio = box_width / float(box_height)
    if source_ratio < box_ratio:
        return int(box_height/float(height) * width), box_height
    else:
        return box_width, int(box_width/float(width) * height)

def test_scaled_size():
    x = scaled_size(width=1871, height=1223, box_width=1920, box_height=1080)
    assert x == (1652, 1080)
    x = scaled_size(width=100, height=100, box_width=1920, box_height=1080)
    assert x ==(1080, 1080)



#File to create windows
#Autor: Ines Caldas and Joel Carneiro

from lib import *
from tkinter.colorchooser import *
import glob
from PIL import ImageTk, Image
import tkinter.ttk as ttk
import tkinter
```

```
#Important Variables
DEFAULT_FONT_TITLE="Arial",10
DEFAULT_WIDTH_CANVASTREE = 38,30

#Function that returns true if num in [x,y]
def valueBetween(num,x,y):
    if (num>=x) and (num<=y): return True
    else: return False

#Icons window
class iconWin:

    #Constructor
    def __init__(self,master, title,icon,type_win,size,
properties=[0,0,0,0,0]):
        self.master = master
        self.root = tkinter.Toplevel(master)

        self.value = 0

        self.root.geometry('%dx%d+%d+%d' % (size[0], size[1],
(self.root.winfo_screenwidth() - size[0])/2,\
                                            (self.root.winfo_screenheight() -
size[1])/2))
        self.root.iconbitmap(bitmap=icon)
        self.root.title(title)
        self.root.minsize(width=size[0], height=size[1])
        self.root.resizable(width=False, height=False)

        if type_win == 'icons':

            F = Frame(self.root)
            F.pack()

            FiguresInsert = Frame(F)
            FiguresInsert.pack()


            files = glob.glob('Data\icons_gui\*')
            i,j = 0,0
            n = 0
            b_lines = [None]*len(files)
            images = [None]*len(files)
            for icon_file in files:
                b_lines[n] = ttk.Button(FiguresInsert,text="Insert Icon", command=lambda icon_file=icon_file:self.sendIcon(icon_file), width=20, style="TButton")
                images[n] = Image.open(icon_file)
                images[n] = images[n].resize((32,32), Image.ANTIALIAS)
                images[n] = ImageTk.PhotoImage(images[n])
                b_lines[n].config(image=images[n])
                b_lines[n].image = images[n]
                b_lines[n].grid(row=j, column=i)

                n = n + 1
                i = i + 1
                if(i > 4):
                    i = 0
                    j = j + 1

            b_cancel = ttk.Button(FiguresInsert, text="Cancel",command=lambda:self.sendIcon(None),width=10)
            b_cancel.grid(columnspan = 5, pady = 10)

    def sendIcon(self, filepath):

        self.value = filepath
        self.root.quit()
        self.root.destroy()

#Clase create windows
class popWin:

    #Constructor
    def __init__(self,title,icon,type_win,size,properties=[0,0,0,0,0]):
        self.root = tkinter.Tk()
        #tkinter.Toplevel(self.root)
        self.value = 0
```

```
                  self.root.geometry('%dx%d+%d+%d'   %   (size[0],   size[1],
(self.root.winfo_screenwidth() - size[0])/2,\
                                             (self.root.winfo_screenheight() -
size[1])/2))
       self.root.iconbitmap(bitmap=icon)
       self.root.title(title)
       self.root.minsize(width=size[0], height=size[1])
       self.root.resizable(width=False, height=False)

       #Weight of Tools
       if type_win=="weight":

Label(self.root,text="Weight",font=DEFAULT_FONT_TITLE,border=10).pack()

           #New thickness line
           F = Frame(self.root)
           F.pack()
           FA = Frame(F)
           FA.pack(side=LEFT)

F_C=Canvas(FA,width=DEFAULT_WIDTH_CANVASTREE[0],height=DEFAULT_WIDTH_CANVASTRE
E[1],bg="white")
           F_C.pack(side=LEFT)

F_C.create_line(0,DEFAULT_WIDTH_CANVASTREE[1]/2+1,DEFAULT_WIDTH_CANVASTREE[0]+
2,DEFAULT_WIDTH_CANVASTREE[1]/2+1,width=1)
           Label(FA,text="  ").pack(side=LEFT)
                                                        Button(FA,text="Weight
1",relief=GROOVE,command=lambda:self.weight(1),width=7).pack(side=LEFT)
           Label(FA,text="  ").pack(side=LEFT)
           FA = Frame(F)
           FA.pack()

F_C=Canvas(FA,width=DEFAULT_WIDTH_CANVASTREE[0],height=DEFAULT_WIDTH_CANVASTRE
E[1],bg="white")
           F_C.pack(side=LEFT)

F_C.create_line(0,DEFAULT_WIDTH_CANVASTREE[1]/2+1,DEFAULT_WIDTH_CANVASTREE[0]+
2,DEFAULT_WIDTH_CANVASTREE[1]/2+1,width=2)
```

```
            Label(FA,text="  ").pack(side=LEFT)
            Button(FA,text="Weight 2",relief=GROOVE,command=lambda:self.weight(2),width=7).pack()

            #New thickness line
            F = Frame(self.root)
            F.pack()
            FA = Frame(F)
            FA.pack(side=LEFT)
            F_C=Canvas(FA,width=DEFAULT_WIDTH_CANVASTREE[0],height=DEFAULT_WIDTH_CANVASTREE[1],bg="white")
            F_C.pack(side=LEFT)
            F_C.create_line(0,DEFAULT_WIDTH_CANVASTREE[1]/2+1,DEFAULT_WIDTH_CANVASTREE[0]+2,DEFAULT_WIDTH_CANVASTREE[1]/2+1,width=3)
            Label(FA,text="  ").pack(side=LEFT)
            Button(FA,text="Weight 3",relief=GROOVE,command=lambda:self.weight(3),width=7).pack(side=LEFT)
            Label(FA,text="  ").pack(side=LEFT)
            FA = Frame(F)
            FA.pack()
            F_C=Canvas(FA,width=DEFAULT_WIDTH_CANVASTREE[0],height=DEFAULT_WIDTH_CANVASTREE[1],bg="white")
            F_C.pack(side=LEFT)
            F_C.create_line(0,DEFAULT_WIDTH_CANVASTREE[1]/2+1,DEFAULT_WIDTH_CANVASTREE[0]+2,DEFAULT_WIDTH_CANVASTREE[1]/2+1,width=4)
            Label(FA,text="  ").pack(side=LEFT)
            Button(FA,text="Weight 4",relief=GROOVE,command=lambda:self.weight(4),width=7).pack()

            #New thickness line
            F = Frame(self.root)
            F.pack()
            FA = Frame(F)
            FA.pack(side=LEFT)
            F_C=Canvas(FA,width=DEFAULT_WIDTH_CANVASTREE[0],height=DEFAULT_WIDTH_CANVASTREE[1],bg="white")
            F_C.pack(side=LEFT)
            F_C.create_line(0,DEFAULT_WIDTH_CANVASTREE[1]/2+1,DEFAULT_WIDTH_CANVASTREE[0]+2,DEFAULT_WIDTH_CANVASTREE[1]/2+1,width=5)
            Label(FA,text=" ").pack(side=LEFT)
            Button(FA,text="Weight 5",relief=GROOVE,command=lambda:self.weight(5),width=7).pack(side=LEFT)
            Label(FA,text=" ").pack(side=LEFT)
            FA = Frame(F)
            FA.pack()
            F_C=Canvas(FA,width=DEFAULT_WIDTH_CANVASTREE[0],height=DEFAULT_WIDTH_CANVASTREE[1],bg="white")
            F_C.pack(side=LEFT)
            F_C.create_line(0,DEFAULT_WIDTH_CANVASTREE[1]/2+1,DEFAULT_WIDTH_CANVASTREE[0]+2,DEFAULT_WIDTH_CANVASTREE[1]/2+1,width=6)
            Label(FA,text=" ").pack(side=LEFT)
            Button(FA,text="Weight 6",relief=GROOVE,command=lambda:self.weight(6),width=7).pack()

            #New thickness line
            F = Frame(self.root)
            F.pack()
            FA = Frame(F)
            FA.pack(side=LEFT)
            F_C=Canvas(FA,width=DEFAULT_WIDTH_CANVASTREE[0],height=DEFAULT_WIDTH_CANVASTREE[1],bg="white")
            F_C.pack(side=LEFT)
            F_C.create_line(0,DEFAULT_WIDTH_CANVASTREE[1]/2+1,DEFAULT_WIDTH_CANVASTREE[0]+2,DEFAULT_WIDTH_CANVASTREE[1]/2+1,width=7)
            Label(FA,text=" ").pack(side=LEFT)
            Button(FA,text="Weight 7",relief=GROOVE,command=lambda:self.weight(7),width=7).pack(side=LEFT)
            Label(FA,text=" ").pack(side=LEFT)
```

```
           FA = Frame(F)
           FA.pack()

F_C=Canvas(FA,width=DEFAULT_WIDTH_CANVASTREE[0],height=DEFAULT_WIDTH_CANVASTRE
E[1],bg="white")
           F_C.pack(side=LEFT)

F_C.create_line(0,DEFAULT_WIDTH_CANVASTREE[1]/2+1,DEFAULT_WIDTH_CANVASTREE[0]+
2,DEFAULT_WIDTH_CANVASTREE[1]/2+1,width=8)
           Label(FA,text=" ").pack(side=LEFT)
                                                        Button(FA,text="Weight
8",relief=GROOVE,command=lambda:self.weight(8),width=7).pack()

           #New thickness line
           F = Frame(self.root)
           F.pack()
           FA = Frame(F)
           FA.pack(side=LEFT)

F_C=Canvas(FA,width=DEFAULT_WIDTH_CANVASTREE[0],height=DEFAULT_WIDTH_CANVASTRE
E[1],bg="white")
           F_C.pack(side=LEFT)

F_C.create_line(0,DEFAULT_WIDTH_CANVASTREE[1]/2+1,DEFAULT_WIDTH_CANVASTREE[0]+
2,DEFAULT_WIDTH_CANVASTREE[1]/2+1,width=9)
           Label(FA,text=" ").pack(side=LEFT)
                                                        Button(FA,text="Weight
9",relief=GROOVE,command=lambda:self.weight(9),width=7).pack(side=LEFT)
           Label(FA,text=" ").pack(side=LEFT)
           FA = Frame(F)
           FA.pack()

F_C=Canvas(FA,width=DEFAULT_WIDTH_CANVASTREE[0],height=DEFAULT_WIDTH_CANVASTRE
E[1],bg="white")
           F_C.pack(side=LEFT)

F_C.create_line(0,DEFAULT_WIDTH_CANVASTREE[1]/2+1,DEFAULT_WIDTH_CANVASTREE[0]+
2,DEFAULT_WIDTH_CANVASTREE[1]/2+1,width=10)
           Label(FA,text=" ").pack(side=LEFT)
```

```
                                                        Button(FA,text="Weight
10",relief=GROOVE,command=lambda:self.weight(10),width=7).pack()

           #New thickness line
           F = Frame(self.root)
           F.pack()
           FA = Frame(F)
           FA.pack(side=LEFT)

F_C=Canvas(FA,width=DEFAULT_WIDTH_CANVASTREE[0],height=DEFAULT_WIDTH_CANVASTRE
E[1],bg="white")
           F_C.pack(side=LEFT)

F_C.create_line(0,DEFAULT_WIDTH_CANVASTREE[1]/2+1,DEFAULT_WIDTH_CANVASTREE[0]+
2,DEFAULT_WIDTH_CANVASTREE[1]/2+1,width=11)
           Label(FA,text="  ").pack(side=LEFT)
                                                        Button(FA,text="Weight
11",relief=GROOVE,command=lambda:self.weight(11),width=7).pack(side=LEFT)
           Label(FA,text="  ").pack(side=LEFT)
           FA = Frame(F)
           FA.pack()

F_C=Canvas(FA,width=DEFAULT_WIDTH_CANVASTREE[0],height=DEFAULT_WIDTH_CANVASTRE
E[1],bg="white")
           F_C.pack(side=LEFT)

F_C.create_line(0,DEFAULT_WIDTH_CANVASTREE[1]/2+1,DEFAULT_WIDTH_CANVASTREE[0]+
2,DEFAULT_WIDTH_CANVASTREE[1]/2+1,width=12)
           Label(FA,text="  ").pack(side=LEFT)
                                                        Button(FA,text="Weight
12",relief=GROOVE,command=lambda:self.weight(12),width=7).pack()

           #New thickness line
           F = Frame(self.root)
           F.pack()
           FA = Frame(F)
           FA.pack(side=LEFT)

F_C=Canvas(FA,width=DEFAULT_WIDTH_CANVASTREE[0],height=DEFAULT_WIDTH_CANVASTRE
E[1],bg="white")
```

```
        F_C.pack(side=LEFT)
        F_C.create_line(0,DEFAULT_WIDTH_CANVASTREE[1]/2+1,DEFAULT_WIDTH_CANVASTREE[0]+2,DEFAULT_WIDTH_CANVASTREE[1]/2+1,width=13)
        Label(FA,text=" ").pack(side=LEFT)
        Button(FA,text="Weight 13",relief=GROOVE,command=lambda:self.weight(13),width=7).pack(side=LEFT)
        Label(FA,text=" ").pack(side=LEFT)
        FA = Frame(F)
        FA.pack()
        F_C=Canvas(FA,width=DEFAULT_WIDTH_CANVASTREE[0],height=DEFAULT_WIDTH_CANVASTREE[1],bg="white")
        F_C.pack(side=LEFT)
        F_C.create_line(0,DEFAULT_WIDTH_CANVASTREE[1]/2+2,DEFAULT_WIDTH_CANVASTREE[0]+2,DEFAULT_WIDTH_CANVASTREE[1]/2+1,width=14)
        Label(FA,text=" ").pack(side=LEFT)
        Button(FA,text="Weight 14",relief=GROOVE,command=lambda:self.weight(14),width=7).pack()

        #New thickness line
        F = Frame(self.root)
        F.pack()
        FA = Frame(F)
        FA.pack(side=LEFT)
        F_C=Canvas(FA,width=DEFAULT_WIDTH_CANVASTREE[0],height=DEFAULT_WIDTH_CANVASTREE[1],bg="white")
        F_C.pack(side=LEFT)
        F_C.create_line(0,DEFAULT_WIDTH_CANVASTREE[1]/2+1,DEFAULT_WIDTH_CANVASTREE[0]+2,DEFAULT_WIDTH_CANVASTREE[1]/2+1,width=15)
        Label(FA,text=" ").pack(side=LEFT)
        Button(FA,text="Weight 15",relief=GROOVE,command=lambda:self.weight(15),width=7).pack(side=LEFT)
        Label(FA,text=" ").pack(side=LEFT)
        FA = Frame(F)
        FA.pack()
```

```
F_C=Canvas(FA,width=DEFAULT_WIDTH_CANVASTREE[0],height=DEFAULT_WIDTH_CANVASTRE
E[1],bg="white")
           F_C.pack(side=LEFT)

F_C.create_line(0,DEFAULT_WIDTH_CANVASTREE[1]/2+2,DEFAULT_WIDTH_CANVASTREE[0]+
2,DEFAULT_WIDTH_CANVASTREE[1]/2+1,width=16)
           Label(FA,text="  ").pack(side=LEFT)
                                                        Button(FA,text="Weight
16",relief=GROOVE,command=lambda:self.weight(16),width=7).pack()

           #New thickness line
           F = Frame(self.root)
           F.pack()
           FA = Frame(F)
           FA.pack(side=LEFT)

F_C=Canvas(FA,width=DEFAULT_WIDTH_CANVASTREE[0],height=DEFAULT_WIDTH_CANVASTRE
E[1],bg="white")
           F_C.pack(side=LEFT)

F_C.create_line(0,DEFAULT_WIDTH_CANVASTREE[1]/2+1,DEFAULT_WIDTH_CANVASTREE[0]+
2,DEFAULT_WIDTH_CANVASTREE[1]/2+1,width=17)
           Label(FA,text="  ").pack(side=LEFT)
                                                        Button(FA,text="Weight
17",relief=GROOVE,command=lambda:self.weight(17),width=7).pack(side=LEFT)
           Label(FA,text="  ").pack(side=LEFT)
           FA = Frame(F)
           FA.pack()

F_C=Canvas(FA,width=DEFAULT_WIDTH_CANVASTREE[0],height=DEFAULT_WIDTH_CANVASTRE
E[1],bg="white")
           F_C.pack(side=LEFT)

F_C.create_line(0,DEFAULT_WIDTH_CANVASTREE[1]/2+2,DEFAULT_WIDTH_CANVASTREE[0]+
2,DEFAULT_WIDTH_CANVASTREE[1]/2+1,width=18)
           Label(FA,text="  ").pack(side=LEFT)
                                                        Button(FA,text="Weight
18",relief=GROOVE,command=lambda:self.weight(18),width=7).pack()
```

```
            #New thickness line
            F = Frame(self.root)
            F.pack()
            FA = Frame(F)
            FA.pack(side=LEFT)
            F_C=Canvas(FA,width=DEFAULT_WIDTH_CANVASTREE[0],height=DEFAULT_WIDTH_CANVASTREE[1],bg="white")
            F_C.pack(side=LEFT)
            F_C.create_line(0,DEFAULT_WIDTH_CANVASTREE[1]/2+1,DEFAULT_WIDTH_CANVASTREE[0]+2,DEFAULT_WIDTH_CANVASTREE[1]/2+1,width=19)
            Label(FA,text=" ").pack(side=LEFT)
            Button(FA,text="Weight 19",relief=GROOVE,command=lambda:self.weight(19),width=7).pack(side=LEFT)
            Label(FA,text=" ").pack(side=LEFT)
            FA = Frame(F)
            FA.pack()
            F_C=Canvas(FA,width=DEFAULT_WIDTH_CANVASTREE[0],height=DEFAULT_WIDTH_CANVASTREE[1],bg="white")
            F_C.pack(side=LEFT)
            F_C.create_line(0,DEFAULT_WIDTH_CANVASTREE[1]/2+2,DEFAULT_WIDTH_CANVASTREE[0]+2,DEFAULT_WIDTH_CANVASTREE[1]/2+1,width=20)
            Label(FA,text=" ").pack(side=LEFT)
            Button(FA,text="Weight 20",relief=GROOVE,command=lambda:self.weight(20),width=7).pack()

            #New thickness line
            F = Frame(self.root)
            F.pack()
            FA = Frame(F)
            FA.pack(side=LEFT)
            F_C=Canvas(FA,width=DEFAULT_WIDTH_CANVASTREE[0],height=DEFAULT_WIDTH_CANVASTREE[1],bg="white")
            F_C.pack(side=LEFT)
```

```
F_C.create_line(0,DEFAULT_WIDTH_CANVASTREE[1]/2+1,DEFAULT_WIDTH_CANVASTREE[0]+
2,DEFAULT_WIDTH_CANVASTREE[1]/2+1,width=21)
           Label(FA,text=" ").pack(side=LEFT)
                                                        Button(FA,text="Weight
21",relief=GROOVE,command=lambda:self.weight(21),width=7).pack(side=LEFT)
           Label(FA,text=" ").pack(side=LEFT)
           FA = Frame(F)
           FA.pack()

F_C=Canvas(FA,width=DEFAULT_WIDTH_CANVASTREE[0],height=DEFAULT_WIDTH_CANVASTRE
E[1],bg="white")
           F_C.pack(side=LEFT)

F_C.create_line(0,DEFAULT_WIDTH_CANVASTREE[1]/2+2,DEFAULT_WIDTH_CANVASTREE[0]+
2,DEFAULT_WIDTH_CANVASTREE[1]/2+1,width=22)
           Label(FA,text=" ").pack(side=LEFT)
                                                        Button(FA,text="Weight
22",relief=GROOVE,command=lambda:self.weight(22),width=7).pack()

           #New thickness line
           F = Frame(self.root)
           F.pack()
           FA = Frame(F)
           FA.pack(side=LEFT)

F_C=Canvas(FA,width=DEFAULT_WIDTH_CANVASTREE[0],height=DEFAULT_WIDTH_CANVASTRE
E[1],bg="white")
           F_C.pack(side=LEFT)

F_C.create_line(0,DEFAULT_WIDTH_CANVASTREE[1]/2+1,DEFAULT_WIDTH_CANVASTREE[0]+
2,DEFAULT_WIDTH_CANVASTREE[1]/2+1,width=23)
           Label(FA,text=" ").pack(side=LEFT)
                                                        Button(FA,text="Weight
23",relief=GROOVE,command=lambda:self.weight(23),width=7).pack(side=LEFT)
           Label(FA,text=" ").pack(side=LEFT)
           FA = Frame(F)
           FA.pack()
```

```
F_C=Canvas(FA,width=DEFAULT_WIDTH_CANVASTREE[0],height=DEFAULT_WIDTH_CANVASTRE
E[1],bg="white")
           F_C.pack(side=LEFT)

F_C.create_line(0,DEFAULT_WIDTH_CANVASTREE[1]/2+2,DEFAULT_WIDTH_CANVASTREE[0]+
2,DEFAULT_WIDTH_CANVASTREE[1]/2+1,width=24)
           Label(FA,text=" ").pack(side=LEFT)
                                                        Button(FA,text="Weight
24",relief=GROOVE,command=lambda:self.weight(24),width=7).pack()

       #Menu insert text
       if type_win=="inserttext":
                                                  Label(self.root,text="Insert
Text",font=DEFAULT_FONT_TITLE,border=10).pack()
           self.texto = Entry(self.root)
           self.texto.pack()
           Label(self.root,text=" ").pack()
                                                             Button(self.root,
text="Write",command=self.sendText,width=10,relief=GROOVE).pack()
           self.texto.focus_force()

       #Menu savefile
       if type_win=="savefile":
                                                  Label(self.root,text="Choose
name",font=DEFAULT_FONT_TITLE,border=10).pack()
           self.texto = Entry(self.root)
           self.texto.pack()
           Label(self.root,text=" ").pack()
                                                             Button(self.root,
text="Save",command=self.sendText,width=10,relief=GROOVE).pack()
           self.texto.focus_force()

       #Menu save
       if type_win=="save":
           lib("sonido","alerta")

Label(self.root,text="Save?",font=DEFAULT_FONT_TITLE,border=10).pack()
           F = Frame(self.root)
           F.pack()
```

```
            Button(F, text="Yes",command=lambda:self.response("yes"),width=5,relief=GROOVE).pack(side=LEFT)
            Label(F, text=" ").pack(side=LEFT)
            Button(F, text="No",command=lambda:self.response("no"),width=5,relief=GROOVE).pack()

        #About
        if type_win=="about":
            Label(self.root,text="Creators"+properties[0],font=DEFAULT_FONT_TITLE,border=5).pack()
            Label(self.root,text="Version: "+str(properties[1]),font=DEFAULT_FONT_TITLE,border=5).pack()
            Button(self.root, text="Close",command=self.root.destroy).pack()

        #License
        if type_win=="license" or type_win=="changelog" or type_win=="help":
            archive = open(properties[1],"r")
            Yscroll = Scrollbar(self.root)
            Yscroll.pack(side=RIGHT, fill=Y)
            text = Text(self.root,wrap=NONE,
            yscrollcommand=Yscroll.set)
            text.focus_force()
            for i in archive: text.insert(INSERT,i)
            text.pack()
            text.configure(state="disabled")
            Yscroll.config(command=text.yview)
            archive.close()

    #Sends response to gui
    def response(self,resp):
        if resp=="yes": self.value = True
        if resp=="no": self.value = False
        self.root.destroy()
    #Enviar un texto
    def sendText(self):
        text = self.texto.get()
        if len(text)>0:
            self.value=text
            self.root.destroy()

    #Send weightes
    def weight(self,weightLine):
        self.value = weightLine
        self.root.destroy()


BACKGROUND_WINDOW = '#404552'
CANVAS_COLOR = '#383C4A'
BUTTON_COLOR = '#4b5162'
BLUE = '#5294e2'
LIGHT_GRAY = '#d7d9dc'
```