```
import RPi.GPIO as GPIO
from picamera import PiCamera
import argparse
import cv2
import numpy as np
import time
import os
import imutils
from imutils.video import FPS
from threading import Thread
import tkinter as tk
from PIL import Image, ImageTk

class software:

        def __init__(self,window,clk,dt,sw,resolucion=(1280, 480), escala = 1,fps=20):

                self.modo = 9
                self.zoom = 0

                # Pines para la conexion con encoder
                self.clk = clk
                self.dt = dt
                self.sw = sw

                #Parametros de camara
                self.camera = PiCamera(stereo_mode='side-by-side',stereo_decimate=False)
                self.camera.resolution = resolucion
                self.camera.framerate = fps
                self.camera.hflip = False
                self.escala = escala

                #Redimensionamiento de imagen a captar
                cam_ancho = int((resolucion[0]+31)/32)*32
                cam_alto = int((resolucion[1]+15)/16)*16
                self.img_ancho = int (cam_ancho * escala)
                self.img_alto = int (cam_alto * escala)

                self.rawCapture = np.zeros((self.img_alto, self.img_ancho, 4), dtype=np.uint8)
                self.stream = self.camera.capture_continuous(self.rawCapture,format="bgra",
use_video_port=True)
                self.frame = None
                self.stopped = False

                #Creacion de interfaz TK
                self.marco = window
                self.width_canvas = 1920
                self.height_canvas = 1080
                self.interval = 1
                self.canvas = tk.Canvas(self.marco, width=self.width_canvas,
height=self.height_canvas, background="black")
                self.canvas.grid(row=0, column=0)

                #Habilitación de interrupciones de encoder
                GPIO.setwarnings(True)
                GPIO.setmode(GPIO.BCM)
                GPIO.setup(self.clk, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
                GPIO.setup(self.dt, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
                GPIO.setup(self.sw, GPIO.IN, pull_up_down=GPIO.PUD_UP)
                GPIO.add_event_detect(self.clk, GPIO.RISING, callback=self.giro, bouncetime=10)
                GPIO.add_event_detect(self.sw, GPIO.FALLING, callback=self.pulsador, bouncetime=10)

                #Inicio de captura de imagenes por camara
                print("Iniciando camara")
```

```
        self.vs = self.start()
        time.sleep(2.0)

        self.main()

def start(self):

        Thread(target=self.update, args=()).start()
        return self

def update(self):

        for f1 in self.stream:

                #Diccionario util para intercambiar el tipo de procesamiento
                switch_funcion = {
                        1: self.mod1_inversion,
                        2: self.mod2_equalization,
                        3: self.sharp,
                        4: self.sob_bn,
                        5: self.sob_nb,
                        6: self.mod5_otsu_an,
                        7: self.mod6_otsu_na,
                        8: self.bin2,
                        9: self.bin3,
                        10: self.original
                }
                self.num_modos = len(switch_funcion)

                #Aplicamos zoom a la imagen
                f2 = self.zoom_x(f1)

                #Se procesa la imagen con el modo seleccionado
                self.frame = switch_funcion.get(self.modo)(f2)

                if self.stopped:
                        self.stream.close()
                        self.camera.close()
                        return

def read(self):
        return self.frame

def stop(self):
        self.stopped = True

def giro(self, clk):

        self.modo
        time.sleep(0.002)
        estado_clk = GPIO.input(self.clk)
        estado_dt = GPIO.input(self.dt)

        #Si existe un giro en el encoder a favor de las manecillas del reloj
        if (estado_clk == 1) and (estado_dt == 0):
                print("direction -> ", self.modo)
                if self.modo >= self.num_modos:
                        self.modo = 1
                else:
                        self.modo += 1
                while estado_dt == 0:
                        estado_dt = GPIO.input(self.dt)
                while estado_dt == 1:
                        estado_dt = GPIO.input(self.dt)
                return
```

```
                #Si existe un giro en el encoder en contra de las manecillas del reloj
                elif (estado_clk == 1) and (estado_dt == 1):
                        print("direction <- ", self.modo)
                        if self.modo <= 1:
                                self.modo = self.num_modos
                        else:
                                self.modo -= 1
                        while estado_clk == 1:
                                estado_clk = GPIO.input(self.clk)
                        return
                else:
                        return

        def pulsador(self,sw):

                time.sleep(0.002)
                estado_sw = GPIO.input(self.sw)

                #Si se presiona el switch del encoder
                if estado_sw == 0:

                        if self.zoom > 5:
                                self.zoom = 0
                        else:
                                self.zoom += 1

                        print("Pulsador presionado")
                        while estado_sw == 0:
                                estado_sw = GPIO.input(self.sw)
                        return
                else:
                        return

        def zoom_x(self, im):

                #Se aplica zoom digital a la imagen captada
                height, width = im.shape[0], im.shape[1]
                r1 = height/width
                pixeles_horizontal = 50*self.zoom
                pixeles_vertical = round((height - r1*(width - 2*pixeles_horizontal))/2)
                recorte = im[pixeles_vertical:height-pixeles_vertical,pixeles_horizontal:width-
pixeles_horizontal]
                resize = imutils.resize(recorte, width=1280)
                return resize

        def original(self,im):
                return im

        #Modo 1 de inversión de imagen
        def mod1_inversion(self,im):
                gray = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)
                inv = (255-gray)
                return inv

        #Modo 2 de alto contraste
        def mod2_equalization(self,im):

                (B, G, R, A) = cv2.split(im)

                # create a CLAHE object (Arguments are optional).
                clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))
                cl1 = clahe.apply(B)
                cl2 = clahe.apply(G)
                cl3 = clahe.apply(R)
```

```
        merged = cv2.merge([cl1, cl2, cl3])

        return merged

#Modo 3 de fondo de un color y objeto de otro color (blanco/negro)
def mod3_otsu_bn(self,im):

        gray = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)
        blurred = cv2.GaussianBlur(gray, (5,5), 0)
        ret, thresh = cv2.threshold(gray,0,255,cv2.THRESH_BINARY_INV+cv2.THRESH_OTSU)
        kernel = np.ones((3,3),np.uint8)
        opening = cv2.morphologyEx(thresh,cv2.MORPH_OPEN,kernel, iterations = 1)
        #sure_bg = cv2.dilate(opening,kernel,iterations=3)
        #erosion_1 = cv2.erode(thresh, kernel, iterations=2)
        #dilate_1 = cv2.dilate(erosion_1, kernel, iterations=2)

        #res = np.hstack((thresh, dilate_1))
        return opening

#Modo 4
def mod4_otsu_nb(self,im):

        mod3_im = self.mod3_otsu_bn(im)
        inverted_binary = cv2.bitwise_not(mod3_im)
        return inverted_binary

#Modo 5 de fondo de un color y objeto de otro color (amarillo/negro)
def mod5_otsu_an(self,im):

        height, width = im.shape[0], im.shape[1]
        canvas = np.zeros((height, width, 3), dtype = "uint8")
        cv2.rectangle(canvas, (0, 0), (width, height), (255, 255, 0), -1)
        mod3_im = self.mod3_otsu_bn(im)
        bitwise_and = cv2.bitwise_and(canvas, canvas, mask = mod3_im)

        return bitwise_and

#modo de fondo de un color y objeto de otro color (amarillo/negro)
def mod6_otsu_na(self,im):

        height, width = im.shape[0], im.shape[1]
        canvas = np.zeros((height, width, 3), dtype = "uint8")
        cv2.rectangle(canvas, (0, 0), (width, height), (255, 255, 0), -1)
        mascara = self.mod4_otsu_nb(im)
        bitwise_and = cv2.bitwise_and(canvas, canvas, mask = mascara)

        return bitwise_and

#obtencion de contornos por medio de sobel
def sob_bn(self,im):

        gray_1 = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)
        sobelX = cv2.Sobel(gray_1, cv2.CV_64F, 1, 0)
        sobelY = cv2.Sobel(gray_1, cv2.CV_64F, 0, 1)
        sobelX = np.uint8(np.absolute(sobelX))
        sobelY = np.uint8(np.absolute(sobelY))
        sobelCombined = cv2.bitwise_or(sobelX, sobelY)
        #(T, thresh) = cv2.threshold(sobelCombined, 200, 255, cv2.THRESH_BINARY_INV)

        return sobelCombined

def sob_nb(self,im):

        sobel_bn = self.sob_bn(im)
        sobel_bn = cv2.cvtColor(sobel_bn, cv2.COLOR_GRAY2BGR)
```

```
                sobel_nb = self.mod1_inversion(sobel_bn)
                return sobel_nb

        def sharp(self,im):

                eq = self.mod2_equalization(im)
                (B, G, R) = cv2.split(eq)
                kernel = np.array([[-1,-1,-1], [-1,9,-1], [-1,-1,-1]])
                shar_B = cv2.filter2D(B, -1, kernel)
                shar_G = cv2.filter2D(G, -1, kernel)
                shar_R = cv2.filter2D(R, -1, kernel)
                merged = cv2.merge([shar_B, shar_G, shar_R])

                return merged

        #Modo 7 imagen ecualizada con contornos negros
        def mod7_equ_cn(self,im):

                gray = cv2.cvtColor(self.sob(im), cv2.COLOR_GRAY2BGR)
                bitwise_and = cv2.bitwise_and(self.sharp(im),gray)

                return bitwise_and

        def bin2(self, im):

                height, width = im.shape[0], im.shape[1]
                canvas = np.zeros((height, width, 3), dtype = "uint8")
                cv2.rectangle(canvas, (0, 0), (width, height), (255, 255, 0), -1)
                gray = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)
                blurred = cv2.GaussianBlur(gray, (5, 5), 0)
                thresh = cv2.adaptiveThreshold(blurred, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
cv2.THRESH_BINARY_INV, 21, 3)
                kernel = np.ones((3,3),np.uint8)
                thresh= cv2.erode(thresh, kernel, iterations=1)
                #thresh = cv2.dilate(erode_1, kernel, iterations=0)
                res = cv2.bitwise_and(canvas, canvas, mask = thresh)

                return res

        def bin3(self,im):

                height, width, channels = im.shape
                canvas = np.zeros((height, width, 3), dtype = "uint8")
                cv2.rectangle(canvas, (0, 0), (width, height), (255, 255, 0), -1)
                gray = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)
                blurred = cv2.GaussianBlur(gray, (5, 5), 0)
                thresh = cv2.adaptiveThreshold(blurred, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
cv2.THRESH_BINARY, 15, 3)
                thresh = cv2.erode(thresh, None, iterations=1)
                #thresh = cv2.dilate(thresh, None, iterations=2)
                res = cv2.bitwise_and(canvas, canvas, mask = thresh)
                return res

        #Main
        def main(self):

                try:

                        frame = self.vs.read()
                        self.image = Image.fromarray(frame)
                        self.image = ImageTk.PhotoImage(self.image)
                        x_0 = (self.width_canvas - self.img_ancho)/2
                        y_0 = (self.height_canvas - self.img_alto)/2
                        self.canvas.create_image(x_0,y_0,image=self.image, anchor="nw")
                        self.marco.after(self.interval, self.main)
```

```
                except KeyboardInterrupt:
                        self.vs.stop()
                        GPIO.cleanup()

if __name__ == '__main__':

        #Creación de interfaz
        root = tk.Tk()
        root.title("Interfaz gráfica para aplicación VR")

        #Ejecución de interfaz
        software(root,clk=17,dt=18,sw=23)
        root.mainloop()
```