Práctica 1:

Contornos activos cuadrados/rectángulos.

José Estévez Fernández.

30/10/2018

Contenido

INTRODUCCIÓN.

Esta práctica consiste en una aplicación en ***C++/Pyton*** bajo ***Qt*** (facultativamente) con el uso de las librerías ***OpenCV***, capaz de capturar un tren de video de la vida real, procesar adecuadamente los contornos, buscar aquellos que tengan un aspecto similar a un cuadrado o a un rectángulo y contarlos/etiquetarlos de un color en especial en la imagen original.

1.-Tanto el valor de sigma de la gaussiana, como el valor del umbral de ***Canny*** deben tener control de usuario por medio de widgets de ***Qt***.

2.-Uso recomendado de ***DrawContours*** y ***FindContours***.

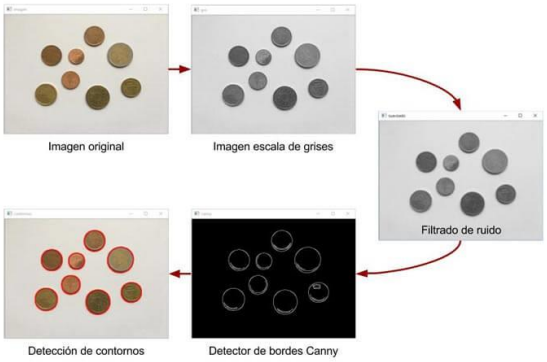
3.-Los códigos fuente y la documentación se depositarán en éste aula virtual en las fechas estipuladas.

4.- Se recomienda diseñar e implementar una clase nueva (de ***QMainWindow***) que gestione la búsqueda de contornos cuadrados/rectángulos.

DETECTAR OBJETOS.

Para detectar objetos es necesario seguir una serie de pasos:

1. Capturar imagen.
2. Convertir la imagen a escala de grises.
3. Filtrar la imagen para eliminar el ruido.
4. Aplicar el detector de bordes ***Canny***.
5. Buscar los contornos dentro de los bordes detectados.
6. Dibujar dichos contornos.



Cada una de estas fases dará como resultado una imagen que será la entrada de la siguiente fase.

CAPTURAR IMAGEN.

Al tratarse de un tren de vídeo el proceso de captura de imagen se realiza mediante un bucle infinito hasta que se pulsa la tecla "q". Esto se lleva a cabo con las siguientes instrucciones:

```
4  # Inicia la captura de vídeo
5  video = cv2.VideoCapture(0)
6
7  # Bucle para procesar el video capturado
8  while True:
9
10     check, frame = video.read()
11
12     key = cv2.waitKey(1)
13
14     if key==ord('q'):
15         break
16
17 video.release()
18 cv2.destroyAllWindows()
```

Al pulsar "q" se produce a liberar la captura de vídeo y a eliminar las pantallas abiertas.

CONVERTIR LA IMAGEN A ESCALA DE GRISES.

Trabajar con imágenes a color tiene un coste computacional exponencial, por ejemplo, el espacio de color ***RGB*** tiene tres componentes de color (R de Red, G de Green y B de Blue) por lo que es como si se trabajase con tres imágenes diferentes, una para cada color.

Además para aplicar el detector de bordes del paso cuarto es necesario que la imagen esté en escala de grises.

Para ello utilizamos la función **cvtColor** de ***OpenCV***:

```
12     # Convertimos a escala de grises
13     gris = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

Donde:

- **gris:** Es la imagen resultante en el nuevo espacio de color.
- **frame:** La imagen capturada en el proceso anterior.
- **cv2.COLOR_BGR2GRAY (tipo de conversión):** Es una constante que indica de qué espacio a qué espacio se va a convertir la imagen.

FILTRADO DEL RUIDO EN UNA IMAGEN.

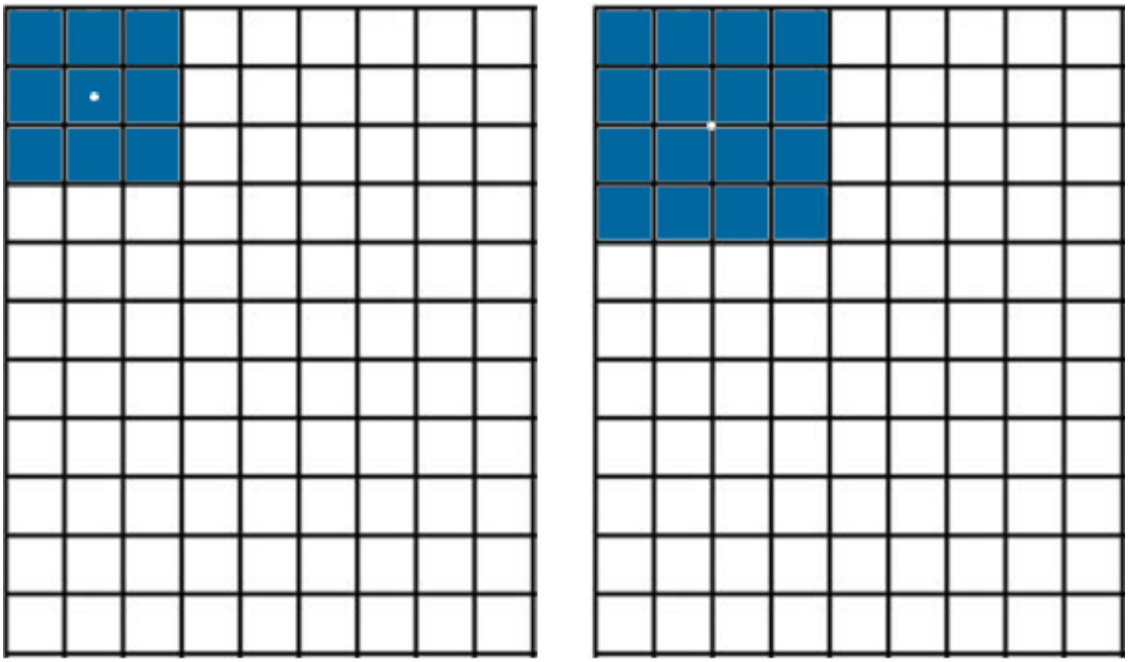
Las **imágenes digitales no son perfectas**. El ruido inherente de los propios dispositivos o los efectos contraproducentes por iluminación alteran la realidad.

Es algo que no solo sucede en los entornos visuales. Cualquier señal digital o analógica está expuesta a diferentes fuentes de ruido. Hay que ser consciente de este problema y saber cómo corregirlo.

En el tratamiento digital de imágenes hay diferentes métodos para eliminar el ruido (promediado, mediana, Gaussiano o bilateral).

Todos utilizan la misma operación matemática, la ***convolución***. Consiste en ir recorriendo píxel a píxel una imagen con una máscara o kernel de $N \times N$. Este tamaño determina el número de píxeles con el que vamos a trabajar.

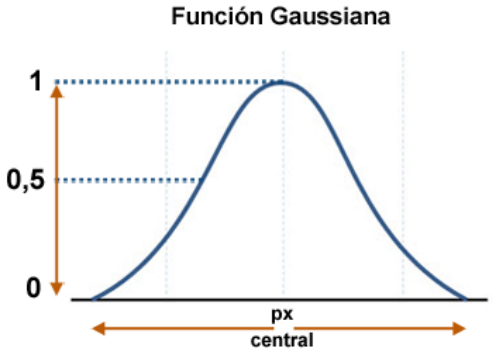
Es muy importante que **el tamaño sea impar para siempre tener un píxel central** que será el píxel en cuestión que estamos tratando.



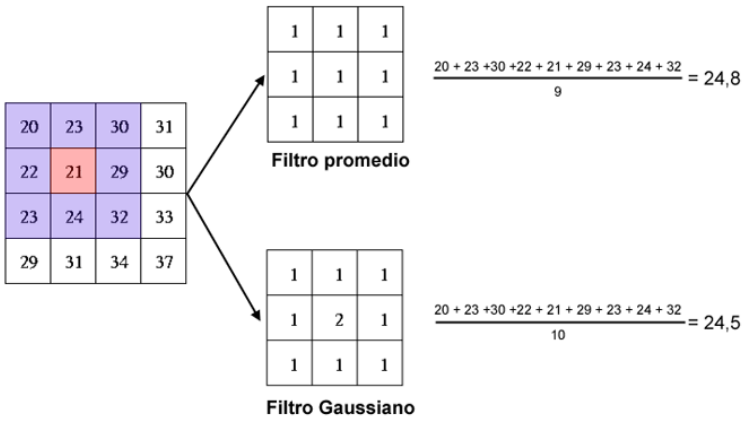
El objetivo es suavizar la imagen es decir, eliminar los detalles. Se puede ver como un desenfoque en una cámara fotográfica. En nuestro caso vamos a utilizar el ***filtrado Gaussiano.***

Filtro Guassiano para la eliminación de ruido en imágenes.

El filtrado Gaussiano es igual que un promediado pero ponderado. Esta ponderación se hace siguiendo la ***Campana de Gauss***. Con esto se consigue dar más importancia a los píxeles que están más cerca del centro de los que están más alejados.



Para poder aplicarlo en una imagen debemos hacerlo en dos dimensiones. Esto lo hacemos a través de una máscara o kernel de convolución.



El filtrado Gaussiano lo hemos realizado mediante el método de OpenCV **GaussianBlur.**

```
17      # Aplicar suavizado Gaussiano
18      gauss = cv2.GaussianBlur(gris, (5,5), 0)
```

Donde:

- **gauss:** Es la imagen desenfocada resultante.
- **gris:** Es la imagen original, la que queremos suavizar o desenfocar.
- **n X n:** es el tamaño del ***kernel*** o máscara de ***convolución***. Debe ser impar.
- **σ (sigma):** Representa la desviación estándar en el eje X, es decir, la anchura de la *campana de Gauss*. El valor **0** indica que será ***OpenCV*** quien se encargará automáticamente de calcular ese valor para el ***kernel*** o máscara que hemos elegido. Esta es la opción aconsejable.

DETECTOR DE BORDES.

El proceso para detectar bordes con Canny se divide en 3 pasos.

- Detección de bordes con Sobel.
- Supresión de píxeles fuera del borde.
- Aplicar umbral por histéresis.

Uno de los grandes inconvenientes de los algoritmos de la visión artificial es la parametrización. Muchas técnicas **requieren establecer parámetros o condiciones iniciales** según cada situación. En muchas ocasiones esos parámetros son exclusivos para una determinada iluminación o perspectiva.

Esto dificulta mucho a la hora de buscar una solución única para diferentes situaciones y por lo tanto, algo que funciona correctamente en unas condiciones de iluminación no tiene por qué funcionar en otras circunstancias.

Detección de bordes con Sobel.

El ***detector de bordes Sobel*** se basa en el cálculo de la primera derivada. Esta operación matemática mide las ***evoluciones*** y ***los cambios de una variable***. Básicamente se centra en ***detectar cambios de intensidad***.

Cuando hablamos de bordes en una imagen, hablamos de los píxeles donde hay un cambio de intensidad.

Filtrado de bordes mediante la supresión non-maximun.

El objetivo en esta fase es poder quedarnos con aquellos bordes que cumplan cierta condición. En el detector de bordes ***Canny*** serán aquellos que tengan como grosor 1.

La supresión ***non-maximun*** es una técnica que permite adelgazar los bordes basándose en el gradiente.

Aplicar umbral por histéresis

La ***umbralización*** de imágenes nos permite también segmentar una imagen en sus diferentes objetos. Básicamente consiste en determinar un umbral por el cual se decide si un píxel forma parte del fondo o forma parte de un objeto.

En el detector de bordes ***Canny*** este es el último paso. Al contrario que el ***umbral simple***, el ***umbral por histéresis*** se centra en establecer dos ***umbrales***, ***uno máximo y otro mínimo***.

Esto ayudará a determinar si un píxel forma parte de un borde o no. Pueden darse 3 casos:

- Si el valor del píxel es mayor que el umbral máximo, el píxel se considera **parte del borde**.
- Un píxel se considera que no es borde si su valor es menor que el umbral mínimo.
- Si está entre el máximo y el mínimo, será parte del borde si está conectado con un píxel que forma ya parte del borde.

Detector de bordes Canny con OpenCV.

Todo este proceso puede ser extremadamente difícil de implementar. Gracias a ***OpenCV*** todo esto resulta relativamente sencillo y práctico.

El método para calcular los bordes con el método ***Canny*** es el siguiente.

```
22      # Detectamos los bordes con Canny
23      canny = cv2.Canny(gauss, 50, 250)
```

Donde:

- **canny:** es la imagen resultante. Aparecerán los bordes detectados tras el proceso.
- **gauss:** es la imagen procesada en el paso anterior.
- **Umbral minimo:** es el umbral mínimo en la ***umbralización por histéresis.***
- **Umbral maximo:** es el umbral máximo en la ***umbralización por histéresis.***

Como ya he comentado, el ***umbral mínimo*** y el ***máximo*** dependerá de cada situación.

BUSCAR LOS CONTORNOS DE UNA IMAGEN.

Hay que saber la diferencia que existe entre un borde y un contorno. Los bordes, como hemos visto anteriormente, son cambios de intensidad pronunciados. Sin embargo, **un contorno es una curva de puntos sin huecos ni saltos**, es decir, tiene un principio y el final de la curva termina en ese principio.

CONTORNO NO CONTORNO

El objetivo de esta fase es analizar todos los bordes detectados y comprobar si son contornos o no. En ***OpenCV*** lo podemos hacer con el método ***findContours***.

```
27      # Buscamos los contornos
28      (_, contornos,_) = cv2.findContours(canny.copy(), cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
```

Donde:

- **Resultado:** obtenemos 3 valores como resultados, el interesante para nosotros es contornos.
    - **imagen:** la imagen después de aplicar la detección de contornos. Es modificada y por lo tanto no podremos utilizarla.
    - **contornos:** es una lista de Python con todos los contornos que ha encontrado.
    - **jerarquía:** la jerarquía de contornos.

- **canny.copy()(imagenbinarizada):** Es la imagen donde se ha detectado los bordes o umbralizado. Este método modifica esta imagen así que es conveniente que sea una

copia. Como norma general, la imagen binarizada debe ser con el fondo negro y los objetos a ser buscados deben ser blancos.

- **modo_contorno:** Es un parámetro interno del algoritmo que indica el tipo de contorno que queremos. Puede tomar diferentes valores ***RETR_EXTERNAL***, ***RETR_LIST***, ***RETR_COMP*** y ***RETR_TREE***.
- **metodo_aproximacion:** Este parámetro indica cómo queremos aproximar el contorno. Básicamente le decimos si queremos almacenar todos los puntos del contorno. Imagínate que tienes una línea recta ¿para qué quieres almacenar todos los puntos? con dos valdría. A lo mejor ahora no le encuentras sentido pero cuando se analizan muchos objetos en imágenes grandes, tendrás que llevar mucho cuidado con el tiempo de procesado y con la memoria. Puede tomar dos valores ***CHAIN_APPROX_NONE*** que toma todos los puntos y ***CHAIN_APPROX_SIMPLE***, que elimina todos los puntos redundantes.

DIBUJAR RECTÁNGULOS/CUADRADOS.

Por último nos queda dibujar los rectángulos/cuadrados que hemos encontrado en la imagen.

Para dibujar los rectángulos/cuadrados es necesario recorrer todos los contornos encontrados, aproximar una curva al contorno para descubrir el número de puntos que conforman el perímetro para saber el polígono que forma. Para esto se utiliza el método ***approxPolyDP.***

```
30      for cntr in contornos:
31          if cv2.contourArea < 8000:
32              continue
33  
34          approx = cv2.approxPolyDP(cntr,0.06*cv2.arcLength(cntr,True),True)
35          if len(approx)==4:
36              print(2)
37              (x,y,w,h) = cv2.boundingRect(cntr)
38              cv2.rectangle(frame, (x,y), (x+w, y+h), (0, 255, 0), 3 )
```

Una vez obtenidos los puntos, sólo pintaremos aquellos que tengan cuatro puntos. Para pintar un rectángulo utilizaremos el método ***boundingRect*** para obtener las coordenadas, ancho y alto del contorno. Y el método ***rectangle*** para pintar un rectángulo en base a los puntos obtenidos con el método anterior.