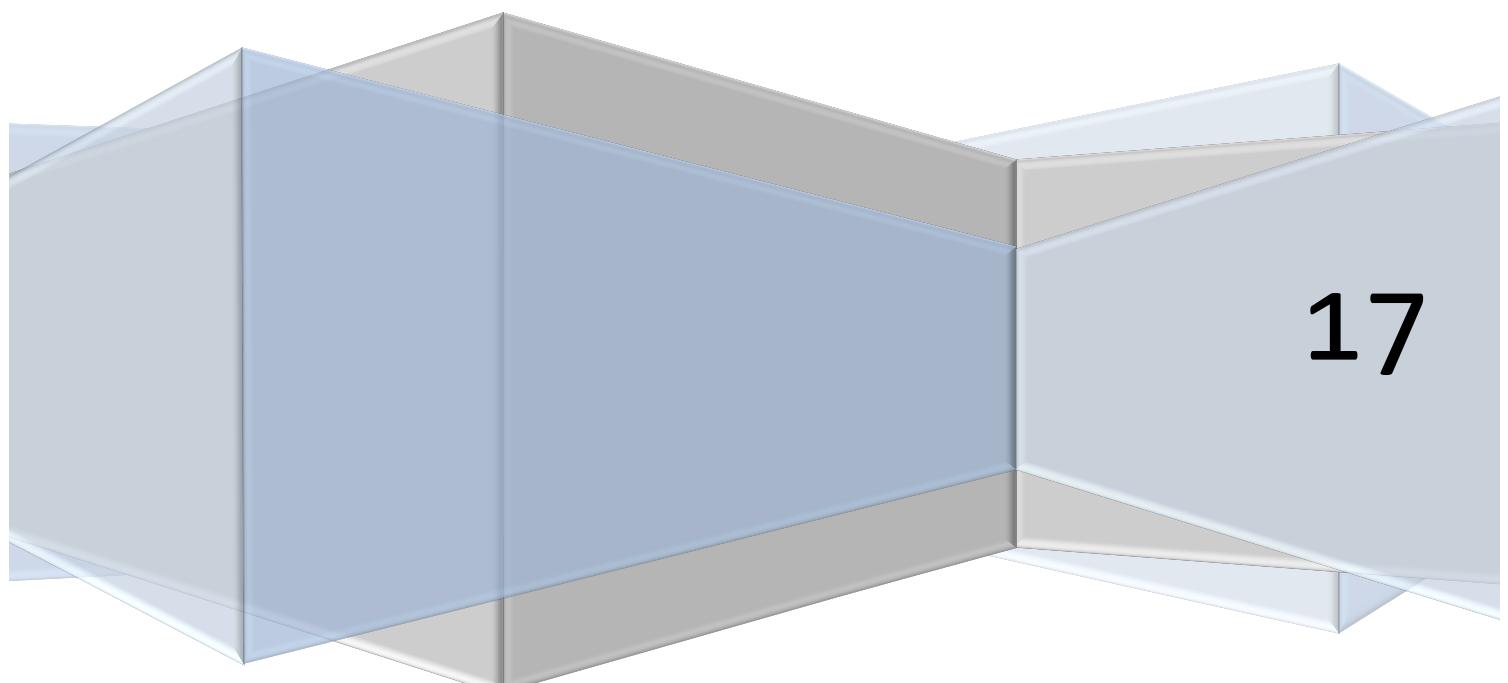


Trabajo 3

CONTORNOS

Jorge Andrés - 679155

Javier Aranda - 679184



INTRODUCCIÓN

El objetivo de este este tercer trabajo es desarrollar mediante “OpenCV” un programa que detecte los contornos de una imagen y sea capaz de, a partir de ellos, calcular el punto de fuga de un pasillo. Para el cálculo de los contornos se han estudiado los operadores de “Sobel”, “Scharr” o “Canny” con un posible filtro Gaussiano previo para reducir el ruido de la imagen.

Se han desarrollado dos programas distintos, uno que es capaz de encontrar el punto de fuga de una imagen pasada como parámetro, y otro que calcula el punto de fuga en tiempo real en las imágenes captadas por la cámara. El primero de ellos se ha desarrollado con los operadores de “Sobel” y “Scharr”, mientras que el segundo usa el operador de “Canny” definido por “OpenCV”.

LECTURA DE IMAGEN

Para la parte obligatoria del trabajo, se ha desarrollado un programa que leyera una cierta imagen de la ruta indicada como parámetro. Estas imágenes, para indicar el modo de lectura al método “imread”, se comprobaron si estaban en blanco y negro o a color. Aunque pareciera algo que no podía tener relevancia, era necesario por si se debía aplicar una conversión de RGB a escala de grises, ya que el mejor método para la búsqueda de contornos es utilizar una imagen en escala de grises.

Una vez se leía la imagen, se comprobaba si se había conseguido con éxito o, por el contrario, si no existía y había ocurrido un error, se abortaba el programa indicándolo. Si se había podido leer la imagen correctamente, se mostraba por pantalla en una ventana para poder ver sobre ella todo el procedimiento que se le fuera aplicando hasta la detección final del punto de fuga.



Imagen a tratar

CÁLCULO DEL GRADIENTE

Tras tener ya la imagen indicada en escala de grises, el siguiente paso era la detección de los puntos de contorno. Para ello, se aplicó en primer lugar un filtro Gaussiano con sigma variable, ya que las máscaras que se iban a utilizar para el cálculo del gradiente de la imagen eran muy sensibles al ruido. Con ello conseguíamos una imagen filtrada que ofrecería mejores resultados finales que aplicando la detección del punto de fuga a la imagen original directamente.

Para aplicar el filtro Gaussiano, se utilizó la función “GaussianBlur” de “OpenCV”. Dicha función recibía como parámetros la imagen original, así como una matriz para guardar la imagen con el filtro aplicado. Además, se le podía indicar el tamaño del “kernel”, y en caso de ser 0, la desviación típica en cada eje que sería la que tendría en cuenta. Con este método, ya teníamos la imagen lista para seguir el procedimiento.



Imagen con filtro gaussiano

A partir de la imagen filtrada, se estudiaron los posibles operadores capaces de calcular el gradiente en cada uno de los ejes. Para ello, se miró la documentación de los operadores de “Sobel” y “Scharr”, que eran máscaras que se aplicaban a toda la imagen, y devolvían la aproximación del gradiente de la función de intensidad de una imagen. La diferencia principal de los dos métodos es que “Scharr” aproximaba mejor la primera derivada que la función de “Sobel”.

| | | |
|----|---|---|
| -1 | 0 | 1 |
| -2 | 0 | 2 |
| -1 | 0 | 1 |

| | | |
|----|----|----|
| 1 | 2 | 1 |
| 0 | 0 | 0 |
| -1 | -2 | -1 |

Máscaras de Sobel para gradiente “x” e “y” respectivamente

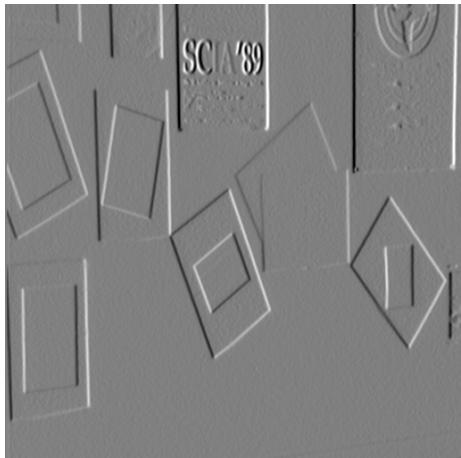
En primer lugar, se utilizó el operador de “Sobel” que recibía como parámetros la imagen filtrada, así como una matriz para devolver el gradiente indicado. Dicho gradiente que nos interesaba era en dirección del eje “X” e “Y” para poder calcular a partir de ellos el ángulo u orientación hacia la que iba la función de iluminación indicada. Para distinguirlos, se realizó la llamada a esa función con los parámetros “0” y “1” en los dos ejes según se quisiera el eje “X” o “Y”. La única particularidad de esta función es que pedía el tipo de dato en el que debía devolver los valores del gradiente y, en nuestro caso, usamos el tipo “CV_32F” ya que era uno de los que aportaba más precisión.

Una vez calculados los gradientes, se probaron a mostrar en dos ventanas para comprobar que el procedimiento se estaba siguiendo satisfactoriamente. Para ello, ya que el rango para mostrar una imagen en escala de grises era [0,255], y el gradiente obtenido estaba en el rango [-255,255] según fuera de claro a más oscuro o viceversa, se aplicó una transformación dividiendo el valor por de cada miembro de la matriz por el escalar 2 y sumándole 255. Con estos sencillos cálculos se conseguía que los valores entraran en el rango esperado.

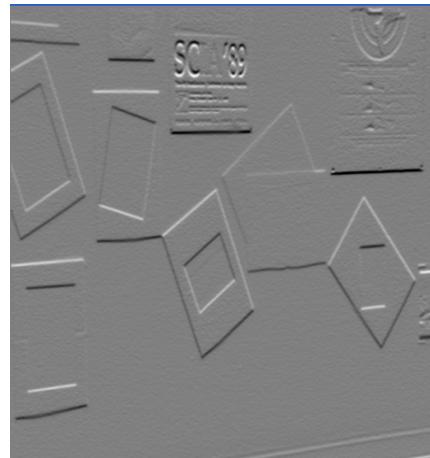
Al mostrar la imagen por pantalla, nos encontramos con que los resultados no eran los esperados, apareciendo cosas que no tenían ningún sentido. Al repasar varias veces el código que llevábamos hasta el momento, no observábamos ningún fallo que pudiera provocar esos resultados, así que miramos la documentación del método “imshow” de “OpenCV” para intentar localizar el fallo. Al observarla detenidamente, nos dimos cuenta que realizaba un procedimiento u otro según fuera el tipo de la matriz que se le pasaba como parámetro.

Si la matriz que recibía era de datos de tipo entero, no realizaba ninguna conversión y los mostraba tal cual, pero si éstos eran de tipo “float”, ya fuera de 32 o 64, lo que hacía era multiplicar los valores por 255 para introducirlos en el rango [0,255]. Al ver los valores que estaban contenidos en nuestra matriz ya estaban en ese rango, nos dimos cuenta que los estaba multiplicando por 255 sin que hubiera necesidad, así que decidimos dividirlos por el escalar 255 para tenerlos en el rango [0,1] y que al dibujarlos lo hiciera correctamente.

Tras conseguir mostrarlos ya como se esperaba, se vio que el gradiente en el eje “X” era como el de las imágenes de prueba proporcionadas, pero, sin embargo, en el eje “Y” los colores estaban cambiados; es decir, donde debía ir blanco era negro y viceversa. Tal y como estaban definidas las funciones de gradiente, el vector gradiente es normal a la superficie apuntando al lado claro, por lo que nuestro eje “Y” estaba invertido a como lo queríamos tener. Para solucionarlo, simplemente se multiplicó la matriz por el valor “-1”, con lo que la imagen mostro los resultados deseados.



Gradiente de X



Gradiente de Y

Con los gradientes ya calculados y comprobado que eran correctos, se pasó a calcular tanto su orientación como su módulo. Para ello, teníamos definidas las siguientes fórmulas que nos permitían obtener ambas magnitudes de una manera sencilla:

$$\text{Módulo} \quad |\nabla f| = \sqrt{(\nabla_x)^2 + (\nabla_y)^2}$$

$$\text{Orientación} \quad \theta = \text{atan} 2(\nabla_y, \nabla_x)$$

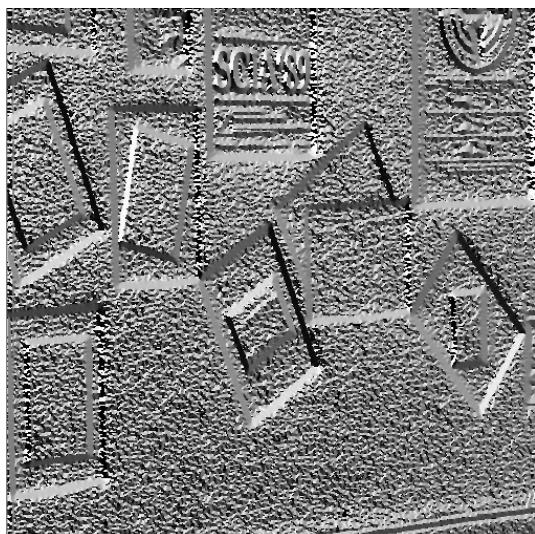
[Formulas del módulo y la orientación](#)

Para calcularlo, se estudió en primer lugar recorrer secuencialmente la matriz de gradiente en eje "X" e "Y" e ir obteniendo los valores, insertándolos en último término en otra matriz que albergaría todos los resultados. Sin embargo, estudiamos la documentación de "OpenCV" por si existía alguna función que ya hiciera lo que esperábamos, viendo que efectivamente sí que estaba ya disponible y era sencilla de usar.

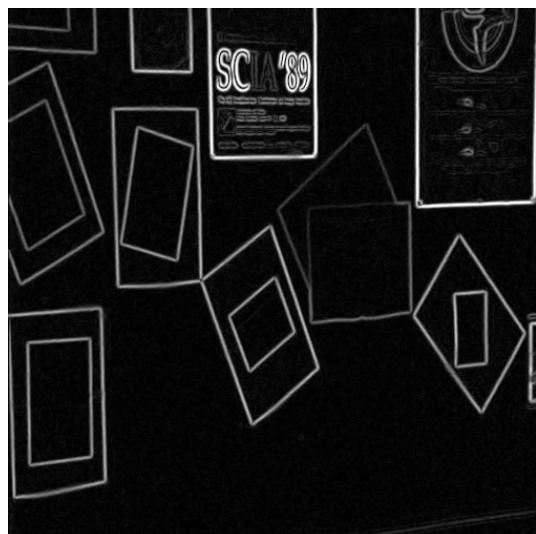
La función se llamaba "cartToPolar" y recibía como parámetros las matrices de los gradientes en ambos ejes, así como las que devolvería los resultados de calcular el módulo y la orientación. Además, aceptaba como parámetro extra si se quería la orientación en grados o radianes, estando por defecto radianes. La única duda que quedaba es en el rango que devolvía el ángulo, ya que lo deseábamos para trabajar con mayor facilidad y mostrarlo en el rango [0,2*PI]. Al mirarlo detenidamente, se vio que lo devolvía justo en ese rango, así que se utilizó y se obtuvieron los resultados.

Para comprobar si eran correctos o no, se realizó el mismo procedimiento que con el gradiente, se probó a mostrarlo en una ventana nueva. El módulo estaba ya en el rango de [0,255] que se debía mostrar en una imagen, simplemente se aplicó la división para dejarlo entre 0 y 1 al tratarse de datos de tipo "float" como se explicó anteriormente. Para el ángulo, era necesario pasarlo a [0,255] para ver todo con más exactitud. Así pues, se dividió entre PI y se multiplicó por 128, consiguiendo el resultado esperado. Finalmente se dividió entre 255 para dejarlo en el rango aceptado para datos "float".

Al mostrarlos por pantalla, se pudo ver que claramente eran muy parecidos a los proporcionados para realizar las comprobaciones de cada paso del procedimiento, y ver que se seguía correctamente. El único punto distinto a destacar era que el módulo salía con tonos muchos más claros que el de las transparencias, algo debido a que se había dividido por algún escalar antes de mostrarlo, por lo que no era ningún problema grave. En lo que al ángulo se refiere, se apreciaba que estaba perfecto en los contornos, aunque como aspecto a destacar eran los contornos verticales. Esos puntos adquirían color blanco o negro indistintamente, algo que pudiera parecer que estaba mal, pero que era correcto ya que el ángulo podía coger valores de [0,10] o [350,360] haciendo cambiar radicalmente el color.



Orientación



Módulo

PUNTOS DE CONTORNO

Tras haber realizado los pasos anteriores exitosamente, se procedió a mostrar los puntos de contorno de la imagen que más adelante permitirán calcular el punto de fuga de la imagen.

Para mostrar los puntos de contorno, únicamente fue necesario recorrer la matriz de la imagen y encontrar los puntos que superaban un cierto umbral definido previamente. Los puntos que superaban dicho umbral eran puntos de contorno. Fue necesario definir dos umbrales diferentes, uno para “Sobel” y otro para “Scharr”, debido a que si se usaba el mismo umbral, los resultados eran muy diferentes para cada operador.



Puntos de contorno de la imagen

TRANSFORMADA DE HOUGH

Hecho esto, se aplicó la transformada de Hough para poder encontrar el punto de fuga de las diferentes imágenes aportadas. Para ello, la transformada encuentra en qué punto se cruzan un mayor número de rectas, siendo este, el punto de fuga. En esta primera parte del apartado, no fue necesario ver el punto en el que se cortaban el mayor número de rectas, sino que tan sólo se miraba el punto de corte con el eje X de la imagen, que estaba localizado en la fila central de la imagen. Así pues, solo hacia falta un vector con tantos elementos como columnas tenía la imagen.

Para cada punto de la imagen cuyo modulo superaba el umbral definido, se obtenía su orientación (calculada previamente). Tampoco se tuvo en cuenta el voto de las líneas verticales y horizontales, ya que estas líneas no intersectan con el punto de fuga y podrían haber empeorado los resultados de la votación. Estas líneas fueron filtradas según un cierto ángulo respecto al eje más cercano.

Para el resto de los puntos, se calculó su rho (distancia del punto al origen), ya que se utilizaban coordenadas polares para el cálculo. Hecho esto, y sabiendo que la coordenada "y" del punto de fuga era igual a 0, se despejó la coordenada "x" en la que la recta del punto de contorno cortaba al punto de fuga. Se realizaron los cálculos precisos y ya se tenía el punto de corte.

Una vez que todos habían votado, se encontró el punto que había recibido el mayor número de votos, siendo este punto la columna en la que se encuentra el punto de fuga de la imagen. Sabiendo esto, y que el punto de fuga se encontraba en la fila central, ya se tenían las coordenadas necesarias para mostrar el punto de fuga en las imágenes.



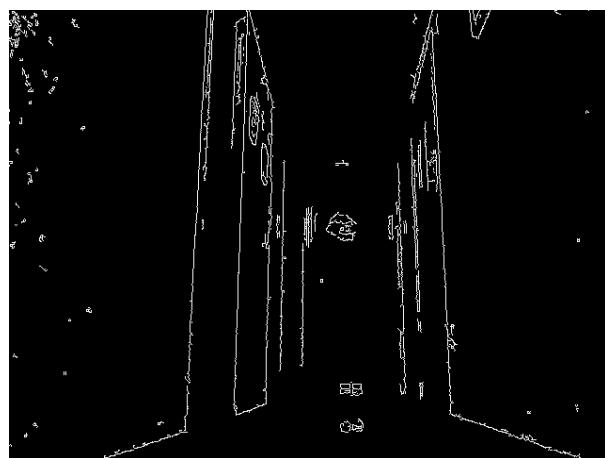
Puntos de fuga de los pasillo 1, 2 y 3 de izquierda a derecho

Como se puede observar en la imagen del primer pasillo, el punto de fuga es erróneo. Esto se debe es que la cámara estaba girada ligeramente al tomar la fotografía, con lo que el punto no se encuentra en la fila central o línea del horizonte. Sin embargo, las otras dos imágenes encuentran el punto de fuga a la perfección.

PUNTO DE FUGA EN TIEMPO REAL

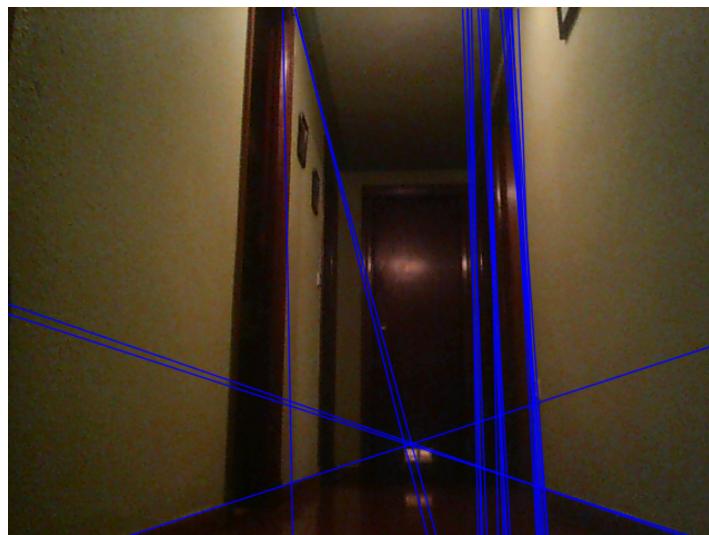
Para la parte opcional, se hizo que el programa procesase imágenes en vivo para obtener el punto de fuga. Para ello se utilizó el algoritmo de “Canny” y la transformada de Hough implementadas ya por “OpenCV” y que ahorraban un gran número de cálculos que se hubieran tenido que hacer a mano.

En primer lugar, en este caso, no fue necesario aplicar un filtro Gaussiano, ya que este ya era aplicado por “Canny” por defecto. Para obtener los contornos de la imagen, se aplicó, como ya se ha dicho, el algoritmo de “Canny”. Esta función recibe como parámetros la imagen en escala de grises sobre la que aplicar el algoritmo, la matriz sobre la que mostrara los resultados, los umbrales que permitirán determinar si una cadena es un contorno (aquella cuyos puntos superen un umbral inferior y al menos uno supere el umbral superior) y el tamaño de la máscara a utilizar.



Contornos obtenidos con Canny

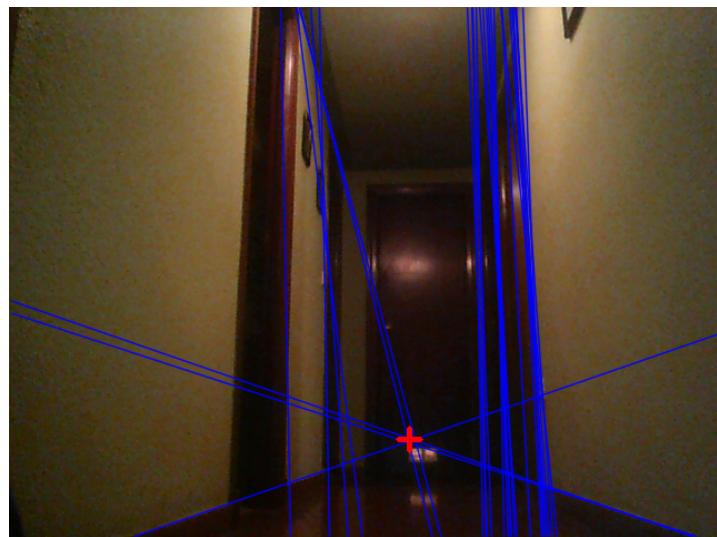
Una vez aplicado el algoritmo de “Canny”, se procedió a aplicar “Hough” para encontrar las rectas de los puntos de contorno que permitirían encontrar el punto de fuga. Para ello, se utilizó la función “HoughLines” que, dada la imagen devuelta por “Canny”, devuelve los rho y las theta de las rectas que podrán votar al punto de fuga. La función “HoughLines” obtiene como parámetros la imagen binaria sobre la que buscar las rectas (en este caso la obtenida con “Canny”), la matriz en la que devolverá los valores rho y theta de las rectas, la división de los parámetros rho y theta para realizar la tabla (siendo en nuestro caso un pixel y un grado respectivamente), y por último, el número de votos mínimo para poder detectar una determinada recta. Al dibujar las rectas en la imagen, se puede, además, intuir donde está el punto de fuga al ver la zona en la que intersectan dichas rectas.



Líneas de contorno obtenidas con Hough

Una vez obtenidas las rectas, se creó una matriz de votación en la que cada línea votaría el punto de fuga de la imagen. Para ello, de nuevo, no se tuvo en cuenta a las líneas horizontales y verticales, ya que no aportaban información útil para este caso. Para la votación, se recorrieron solamente las filas de la imagen, y se comprobó en qué columna cortaba cada recta la imagen, permitiendo así recorrer la matriz en una única dimensión, sumando un voto en cada uno de estos puntos (definir los puntos de la recta sobre la imagen).

Una vez hecha la votación, solo era necesario encontrar el punto más votado, siendo este el punto de fuga. Sin embargo, al funcionar el programa en tiempo real, se producían ciertos fallos al captar las líneas que hacían que el punto de fuga oscilara demasiado en la imagen (una variación de un pixel en una recta hacia que el corte cambiara constantemente). Para solucionarlo, se decidió que cada pixel ocupado por las rectas tan sólo votase a un punto de cada cinco para que así el punto no variase tanto en la imagen final y reducir las oscilaciones. Hecho esto, se obtuvieron unos resultados bastante buenos en los que el punto de fuga apenas variaba, cumpliendo así el objetivo del trabajo.



Punto de fuga obtenido en tiempo real