Proyecto Final: Laboratorio Visión por Ordenador I

Contenido

[1. Introducción 2](#_Toc186217850)

[2. Metodología 2](#_Toc186217851)

[3. Resultados 5](#_Toc186217852)

[4. Futuros desarrollos 5](#_Toc186217853)

# Introducción

A lo largo de la asignatura cuatrimestral hemos dado desde la calibración de una cámara, calculando sus parámetros intrínsecos y extrínsecos con Python, hasta la detección de movimiento y el seguimiento de objetos.

En este proyecto final se desarrolla un proyecto muy completo que consiste en tres apartados: calibración, sistema de seguridad y sistema propuesto, todo a través de una *Raspberry Pi* y una *Camera module 3 WIDE (120º FOV).*

El sistema de seguridad el objetivo consta de la detección correcta de un patrón a través de la extracción de las características, en esta parte se implementa un módulo que pueda detectar líneas, círculos y polígonos. Por otra parte, el descodificador logra memorizar el patrón para determinar si se ha introducido la contraseña correcta, y ceder paso al bloque del sistema propuesto. En este último apartado se inicia el *tracker*, que consta en una simulación de una visión desde un coche, que detectará el movimiento de otros coches (de juguete, a los que la Raspberry Pi grabará e identificará) y los colores de los semáforos.

# Metodología

**Calibración de la cámara.**

Para calibrar la cámara se reutilizó el código de la primer práctica de laboratorio, donde cargamos las imágenes realizadas desde distintos ángulos a un tablero de ajedrez, y detectamos las esquinas interiores gracias a la función *findChessboardCorners*, y acto seguido refinamos esa detección de esquinas con la función *cornerSubPix*. Para obtener un resultado visual dibujamos las esquinas detectadas sobre las imágenes del tablero con la función *drawChessboardCorners* y las guardamos. Hecho esto, utilizamos la función que diseñamos en la primera práctica llamada get\_*chessboard*:points para obtener las coordenadas 3D de las esquinas del patrón y filtramos las esquinas detectadas, con lo que ya podemos calibrar la cámara con *calibrateCamera* y guardar los parámetros resultantes.

**Diagrama de bloques del sistema.**

**Diagrama

Descripción generada automáticamente**

**Secuencia de transformación de la imagen.**

Para la parte del sistema de seguridad hemos transformado la imagen de la siguiente forma:

Picam.capture\_array() -> Frame -> Frame blurred -> Frame BGR to HSV -> Frame segmented by color -> Mask eroded

Para la parte del sistema propuesto, para el tracker:

Picam.capture\_array() -> Frame -> Frame BGR to HSV -> Frame filtered by threshold -> Frame with opening operator applied

Para la parte de la identificación del color del semáforo del sistema propuesto:

Picam.capture\_array() -> Frame -> Frame BGR to HSV -> Frame segmented by color -> Mask eroded -> Mask dilated

**Sistema de seguridad: Detección de patrones y Extracción de información.**

Para la parte del sistema de seguridad se han creado tres módulos: *Figure.py*, *FigureDetector.py* y *Authenticator.py*, cada uno aprovechando funcionalidades de anterior. Con el módulo *Figure.py* se crean las figuras que el sistema va a ser capaz de detectar, en nuestro caso, un cuadrado verde, un círculo azul, un triángulo rojo y un pentágono morado, esta es la contraseña que se memoriza.

El módulo *FigureDetector.py* nos permite identificar individualmente cada una de las figuras anteriores, dependiendo de la figura que se introduzca en el detector. Como cada figura tiene un color diferente, su identificación se ha realizado en primer lugar mediante segmentación de color, quedándonos con la máscara creada para posteriormente erosionar la imagen para eliminar posibles detecciones incorrectas y medir el número de esquinas detectadas con *cv2.findContours* y *cv2.approxPolyDP*. En función del número de esquinas detectadas se clasifica como un polígono u otro. Para el caso del círculo se calcula un coeficiente de circularidad y en función de si se encuentra dentro de unos límites, la figura detectada se clasifica como círculo o no .Además, para comprobar la correcta detección de cada figura se muestra sobre la imagen la figura que se está detectando junto con su nombre (en el caso de detectar alguna figura).

Para la parte de extracción de información, se ha diseñado el módulo *Authenticator.py*. Este módulo segmenta por los distintos colores de cada una de las figuras buscando alguna coincidencia. Si la hay (también en el número de esquinas), se almacena en una lista de patrones detectados, cuya coincidencia con la contraseña se verifica cuando hay 4 elementos en la lista. Si la secuencia de patrones detectados coincide con la contraseña, se muestra un rectángulo en verde alrededor de la imagen de la cámara y se imprime por consola un mensaje indicando que se ha introducido la contraseña correcta durante 2 segundos. En caso contrario, se muestra un rectángulo rojo durante 2 segundos. Transcurridos esos 2 segundos el rectángulo se elimina y se puede volver a registrar otro patrón, hasta que se introduzca la contraseña correcta. Para evitar que se detecte el mismo patrón en una iteración del bucle y en la siguiente, se ha implementado un retraso de 2 segundos desde la detección de un primer patrón, de forma que hasta que no transcurran esos 2 segundos el sistema no detectará ningún otro patrón.

**Sistema propuesto: *tracker*, ampliaciones y salida de vídeo.**

Explicación del algoritmo de Kalman: de los algoritmos que estudiamos en las clases de teoría para el seguimiento de objetos, elegimos Kalman por su manejo ante el ruido y su capacidad predictiva, es un algoritmo clásico muy utilizado. El proceso consiste a propagar y actualizar distribuciones gaussianas y sus covarianzas. Desde el sensor obtenemos la posición del objeto una vez lo hemos detectado, y se aprovecha esa información para manejar el ruido de la imagen y predecir su movimiento a través de modelos de velocidad o aceleración constante.

1. Primero, se introduce el estado inicial (velocidad y posición del objeto inicial) y lo que se trata de lograr, que es predecir el siguiente estado. Para obtener la incertidumbre inicial se usa la covarianza de la gaussiana inicial.
2. Un dibujo de una persona

   Descripción generada automáticamente con confianza bajaImagen que contiene objeto, reloj

   Descripción generada automáticamenteLa siguiente etapa es la de predicción del estado, que consiste en multiplicar el estado anterior por la matriz de transición de estado F (que es el modelo de movimiento) . La actualización de la covarianza se realiza multiplicando esta por la matriz F y sumando el ruido de proceso (error constante que representa el error del modelo, que no es perfecto).

**Figura 1**. Actualización del estado.

**Figura 2.** Actualización de las covarianzas.

1. Por último, pasamos a la etapa de corrección, que consiste en la actualización de estado y una reducción de la incertidumbre. Esto se logra a través del cálculo de distintos parámetros: residual (la innovación) y la ganancia de Kalman (la descripción de la obtención no se incluye en el proyecto ya que no se ha tratado con profundidad, pero la idea principal es corregir las gaussianas). Una vez obtenidos, podemos actualizar el estado y actualizar la covarianza, lo que permite reducir la incertidumbre en los siguientes pasos y obtener una predicción más robusta.

Para la parte del sistema propuesto, hemos decidido hacer un sistema de coche autónomo básico. Para ello, hemos diseñado un *tracker* que siga a un objeto seleccionado por el usuario (en nuestro caso será un coche, pero podría ampliarse a más objetos) y que detecte el color de un semáforo para saber si debe detenerse o si puede continuar. El semáforo solo tiene dos colores: verde y rojo. Para la parte del *tracker* hemos usado una combinación del filtro de Kalman y el algoritmo *meanShift*. Para ello, calculamos la proyección retrospectiva, convirtiendo la imagen a HSV y calculando la probabilidad de que cada píxel pertenezca al objeto basándose en el histograma del objeto seleccionado. A continuación, usamos el algoritmo *meanShift* para encontrar la nueva posición del objeto basándose en la posición retrospectiva. Posteriormente, contando el número de píxeles no negros calculamos cómo de seguro está el sistema de haber encontrado el objeto correcto. Por último, mediante el filtro de Kalman, predecimos dónde debería estar el objeto, corregimos la predicción usando la posición real medida y, si la predicción está muy lejos de la última posición conocida, la ajustamos para evitar pérdidas de seguimiento.

Para la parte de detección del color de un semáforo, hemos usado segmentación de color y, sobre la máscara obtenida, hemos calculado un coeficiente de circularidad que nos indica cuándo un objeto detectado del color de la segmentación es un círculo o no. Para ello, hemos tenido que establecer un intervalo dentro del cual se considerará que el objeto es un círculo. Además, se muestra tanto sobre la imagen como por consola el color detectado.

# Resultados

Haciendo la segmentación por color tanto en el sistema de seguridad como en el sistema propuesto, nos hemos dado cuenta de que el rojo es más complicado de segmentar que otros colores y que el verde es mucho más fácil de segmentar que el resto de los colores utilizados. Aparte de eso, el sistema de seguridad ha terminado funcionando bastante bien, aunque hay alguna vez que detecta erróneamente una figura debido a que la segmentación por color depende en gran medida de la iluminación. Pero, por lo general, la detección de las figuras funciona correctamente.

En la parte del tracker, haciendo pruebas hemos visto que la calidad del seguimiento de la *bounding box* depende en gran medida de la selección inicial del ROI. Si seleccionamos demasiado coche, el seguimiento no es tan bueno que seleccionando una parte lo más uniforme posible del coche. Por último, en la parte de la detección del color del semáforo, nos hemos dado cuenta de que, como comentábamos antes, el rojo es más difícil de detectar, por lo que la detección del semáforo rojo depende en gran medida del ángulo del círculo rojo respecto a la cámara.

# Futuros desarrollos

Como nuestro diseño del *tracker* consiste en la visión desde un coche que claramente podría ser un coche autónomo, entre los futuros desarrollos hemos pensado que podríamos calcular la distancia de la cámara al coche al que estamos haciendo tracking. De forma que, en función de esa distancia, el coche que lleve la cámara pueda saber si debe empezar a frenar, si puede acelerar, etc. Esto lo haríamos con la distancia focal (F) que hemos obtenido al calibrar la cámara y la anchura del objeto (W) que vamos a seguir, ya que tenemos una sola cámara (hay otros métodos disponibles si obtenemos una visión estéreo), y la anchura percibida por la cámara en pixeles (P) que obtenemos al seleccionar el cuadrado que envuelve al objeto cuando determinamos el ROI.

*Dist = (W x F) / P*

De esta manera podríamos imprimir por imagen la distancia a la que está el coche y proporcionar más información. Otros posibles desarrollos en la consecución de este objetivo serían la detección de peatones y otras señales de tráfico más detallas, diseñando de alguna manera un diccionario de las características para poder reconocer correctamente cada una. Por otro lado, se podría implementar sonido, para simular alarmas acústicas cuando la distancia es menor a cierto límite.