Práctica Reconocimiento de Gestos - SIPC

Andrés Concepción Afonso
Brian Samir Santamaría Valero

Introducción

El objetivo de esta práctica es aprender de forma sencilla cómo se procesa una imagen para reconocer los gestos que realizamos, en este caso con una mano. Para ello se usa la librería open source OpenCV, que implementa métodos y funciones tanto para procesar la imagen como para mostrar el resultado del reconocimiento de gestos.

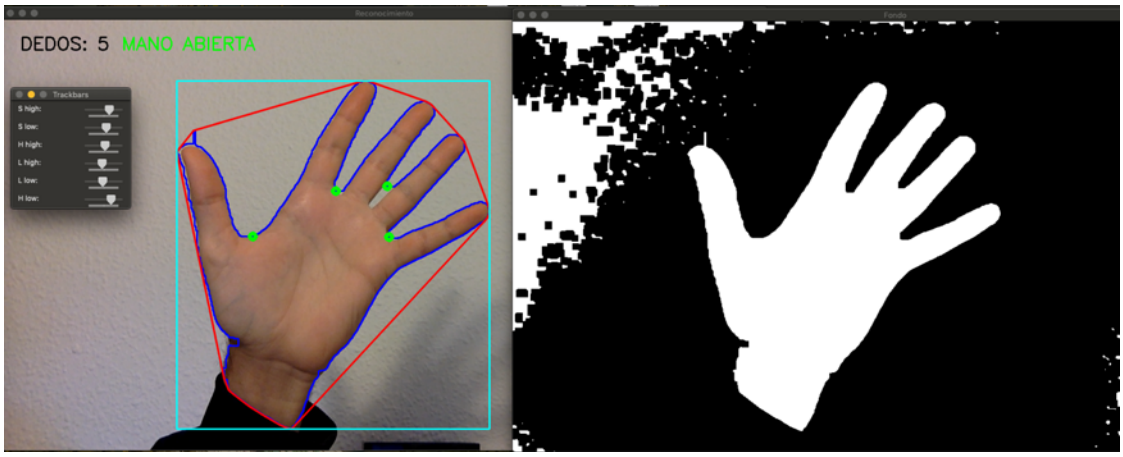
Esta práctica se ha dividido en 4 sesiones de laboratorio, de las cuales dos se realizaron antes de Navidad. Las sesiones se estructuraron de la siguiente forma:

Semana 1

El objetivo de la primera semana fue conseguir separar la mano del resto del fondo de la imagen.

Para ello se usa un método de **sustracción de fondo** (usando el espacio de color HSL) implementado en dos funciones principales. En la primera, `LearnModel()`, se toma una muestra de la superficie a reconocer (la mano), convirtiéndola al espacio de color HLS y tomando una media de los valores de la muestra.

Con esto se genera un modelo de lo que queremos filtrar, que es el que en la segunda función, `ObtainBGMask()`, se compara con la imagen en vivo que recoge la cámara (convirtiéndola también a HLS) y para cada media se genera un umbral binario que denota si los puntos de la imagen están dentro del rango de dicha media o no (que se comprueba con la función `InRange()`. Una vez hecho se suman las imágenes binarias y con esto ya tenemos la máscara de fondo, donde idealmente la mano debería verse blanca y el resto de los elementos del fondo negros.



Interfaz del programa, con la imagen real y la máscara correspondiente (en este caso sin filtrar adecuadamente para mostrar el ruido que se genera en ese caso)

Para ajustar el filtrado de la máscara podemos regular mediante deslizadores los umbrales de **H**(ue), **S**(aturation) y **L**(ightness) para reducir al mínimo el ruido. Se usa el espacio de color HSL en lugar de RGB porque puesto que estamos trabajando sólo con la información de una imagen es más fácil separar los elementos que conforman la misma ya que el color es solo un canal y no tres como en RGB. EL principal problema que nos encontramos es conseguir que la máscara se genere correctamente y que la toma de muestras sea buena, puesto que en caso contrario nos vemos expuestos a más ruido. Dicho ruido se agrava sobre todo si además el fondo no es homogéneo o no hay suficiente contraste entre la mano y el mismo.

Una vez obtenida la máscara debemos refinarla para que se ajuste mejor a la mano, así que usamos distintos métodos para conseguirlo. El primero es aplicar un **filtro de mediana**, `medianBlur()`, que asigna a cada punto la mediana de los valores de los puntos adyacentes (dentro de una zona cuyo tamaño definimos como parámetro), con lo que reducimos el ruido y los falsos positivos (puntos blancos en zonas negras de la máscara o viceversa).

Además, también usamos los **métodos de dilatación y erosión** para suavizar aún más los bordes. Los valores con los que se ajustan estos métodos son dependientes del entorno donde se esté usando la cámara, así que no hay una configuración 100% estándar para dichos métodos. El método de dilatación `dilate()` «engorda» una región, mientras que el de erosión `erode()` la «adelgaza». Esto se consigue extendiendo el valor de los píxeles distintos de cero o iguales a cero, respectivamente, a sus píxeles vecinos en base a una forma geométrica (que en este caso es un rectángulo).

Con esto se consigue que si hay mucha imprecisión en el borde esta se difumine y quede suavizado. De esta manera si por ejemplo dilatamos y luego erosionamos con el mismo factor habremos conseguido una región del mismo tamaño que la original, pero con bordes más suavizados.

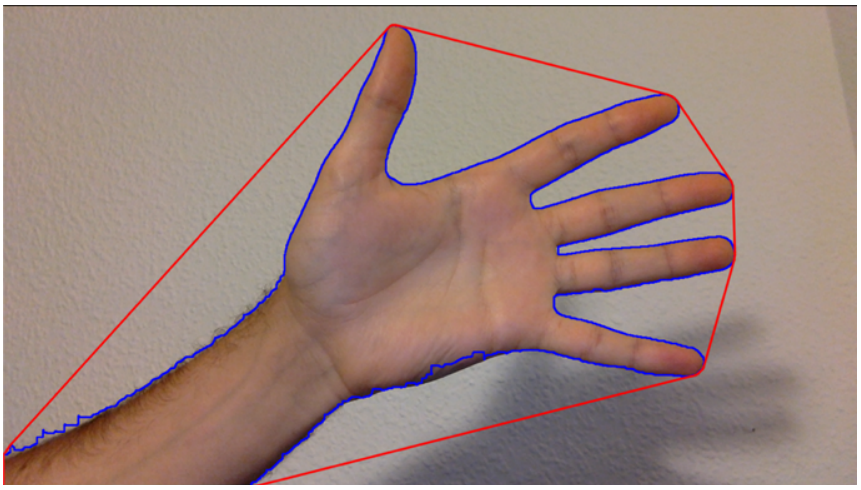
Ejemplo de la mano ya suavizada y con la máscara de fondo aplicada

Semana 2

En la segunda semana se parte de la máscara ya definida y filtrada, así que es hora de conseguir distinguir la mano. Los distintos métodos para conseguirlo se implementan dentro de la función `FeatureDetection()`.

Para ello, lo primero es detectar los contornos de la imagen (que puesto que la misma es binaria B/N es bastante sencillo de hacer, y en un caso ideal si la máscara es perfecta debería detectarse sólo el de nuestra mano). Puesto que no es un caso ideal lo que hacemos es quedarnos con el contorno más largo que se detecte en la imagen (que si hemos tomado bien las muestras sí que debería ser el de la mano).

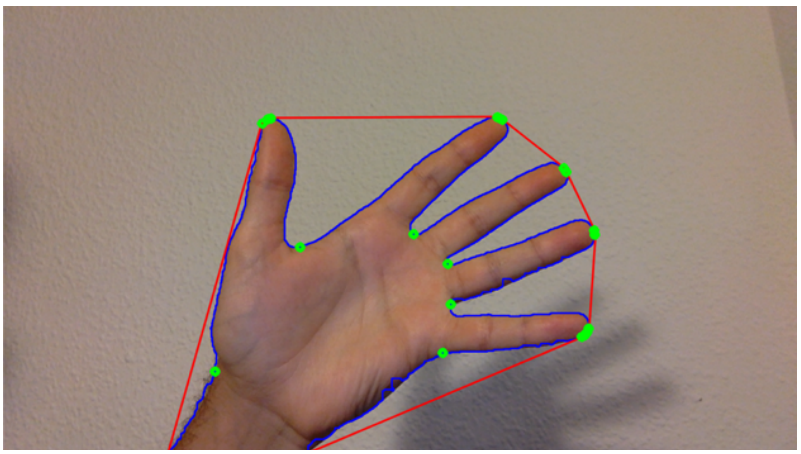
A partir de este contorno generamos la malla convexa **(convex hull)**, un polígono que formalmente se puede definir como el más pequeño que contiene a todos los puntos del contorno. En este caso ello implica que entre los vértices de dicho polígono estarán incluidos los dedos (porque son «extremos relativos» del contorno).



Ejemplo del contorno de la mano (azul) y el convex hull (rojo)

Para poder distinguir cuáles son los correspondientes a los dedos hacemos uso de los defectos de convexidad (**convexity defects**), que son los puntos más «profundos» del contorno con respecto a cada una de las aristas del convex hull. Estos puntos quedan definidos por tres puntos: el de inicio **(s)**, el de fin **(e)** (que corresponden a los dos vértices de cada una de las aristas) y el más lejano **(f)**, que corresponde al punto del defecto de convexidad en sí.

Podemos aprovecharnos de que sabemos que los defectos de convexidad entre los dedos tienen dos características particulares: que el ángulo del vértice **f** en el triángulo que forma con **s** y **e** tiene una apertura máxima reducida, y que **f** está a una distancia alta con respecto a **s** y **e.**



Vemos que muestra todos los defectos de convexidad que encuentra, no solo los de los dedos.

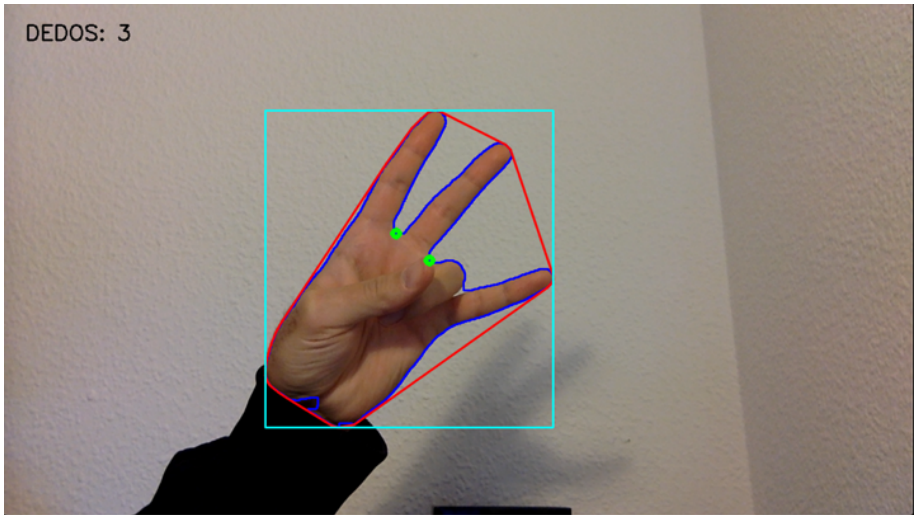
En este caso hemos optado por suponer que los defectos de convexidad son dedos si dicho ángulo es menor que 90°. Sin embargo, no podemos usar como distancia la «altura» en píxeles de la imagen directamente, puesto que si alejamos la mano esta altura sería menor y daría errores. En su lugar lo que hacemos es calcular el «rectángulo mínimo» que contiene al convex hull (`boundingRec()`) y en base a su altura calculamos la distancia mínima para que sea un defecto de convexidad correspondiente a los dedos (nosotros suponemos que lo es si la distancia es mayor a 0.2 * altura del «rectángulo mínimo»)

Semana 3

Una vez implementados correctamente la detección de la mano y los defectos de convexidad podemos empezar a jugar con el reconocimiento de gestos. En este caso, hemos optado por tres aplicaciones muy simples:

- Contar el número de dedos

Consiste simplemente en un contador que muestra por pantalla el número de dedos abiertos. Puesto que se forma un defecto de convexidad con dos dedos extendidos podemos saber que el número de dedos será uno más que el número de defectos de convexidad detectados.

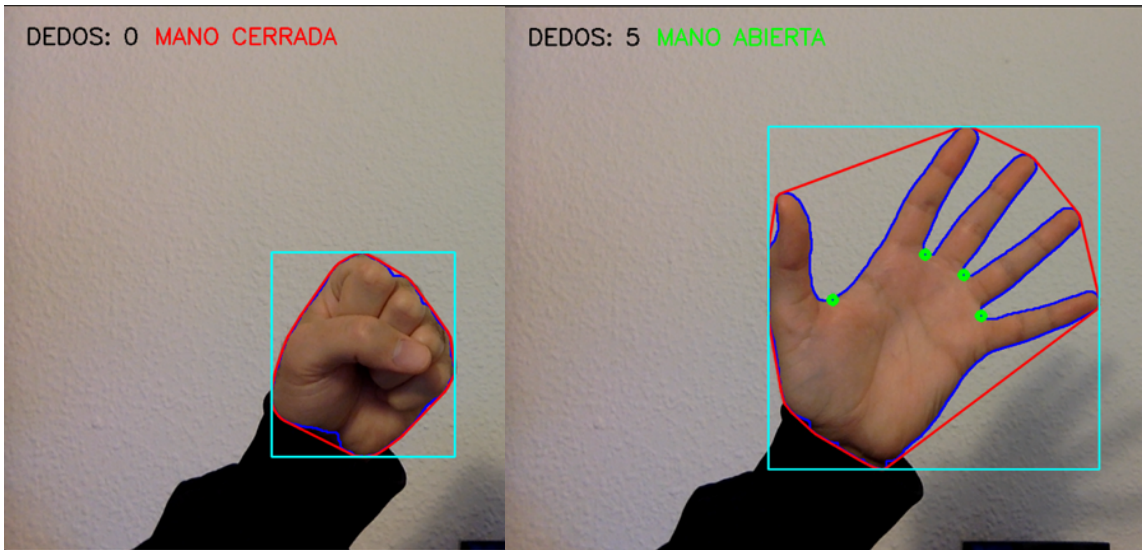


Ejemplo de detección de tres dedos extendidos

Para distinguir entre un dedo extendido y el puño cerrado usamos las dimensiones del rectángulo mínimo, puesto que con un dedo extendido la proporción entre el alto y el ancho es distinta a si el puño está cerrado.

- Mostrar si la mano está abierta o cerrada

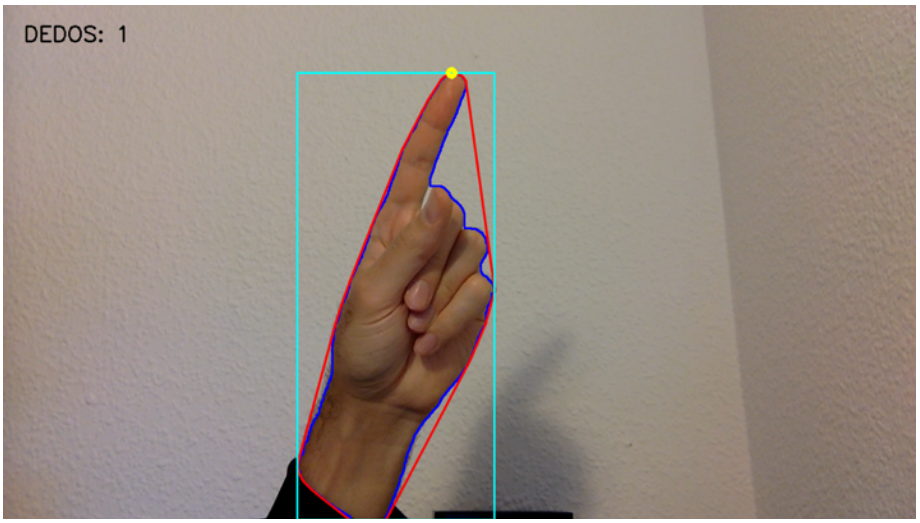
Simplemente una extensión del apartado anterior. Si el número de defectos de convexidad es 4 significa que la mano está totalmente abierta; y si es 0 y además detectamos que la proporción del rectángulo mínimo corresponde a la de ningún dedo, que la mano está cerrada. En cada uno de esos casos se muestra un mensaje indicando si está abierta o cerrada.



Ejemplo de mensajes con la mano cerrada y abierta

- Puntero con un solo dedo

De igual manera, si se detecta que hay un solo dedo extendido dibujamos un círculo que siga a la punta de este. Esto se consigue recorriendo la lista de defectos de convexidad y escogiendo el punto **s** con la coordenada y más elevada (es decir, el punto del convex hull que se encuentra más arriba en la imagen).



Ejemplo del «puntero»

Una vez tenemos esto se podrían implementar otras cosas (como dibujar con dicho dedo), pero por falta de tiempo no hemos podido hacerlo.