**Antonio Raúl Guijarro Contreras Florentín Pérez González Adrián Epifanio Rodríguez Hernández Christian Torres González**

**Sistemas de interacción persona-computador,  Universidad de La Laguna**

**Reconocimiento de gestos con OpenCV**

****

**ÍNDICE**

1. Substracción de fondo ………………………….…………….. 2 - 4
   1. My BG Subtractor Color …………………….…………..…. 2
   2. Obtención de la máscara binaria ……….……………....... 3
   3. Ventajas e inconvenientes del Método ………...………… 4
2. Selección del contorno de la mano …………………………. 5 - 6
   1. Reducción de ruido ……………………...……………….… 5
   2. Generación del contorno de la mano ……..……………… 6
3. Convex Hull ……………………………………………………...… 7
4. Defectos de Convexidad ……………………………………...… 8
5. Contador de dedos………………………………………………... 9
6. Bounding Rect ……………………………….………………..… 10
7. Movimiento de la mano …………………………………………. 11
8. Pintar en la pantalla con el dedo ………………………….. 12
9. **Substracción de fondo**
   1. **Obtención del modelo de la mano**

Para conseguir substraer con éxito el fondo de las imágenes que se captan con la cámara y con objeto de que ello permita captar eficazmente lo que es una mano de lo que no, se ha recurrido a una serie de implementaciones en código que, en su conjunto, permiten tal tarea.

En primera instancia, convertimos las imágenes que se captan a través de la cámara subyacente a HLS ya que los valores de Hue (Matiz), Ligthness (Brillo) y Saturation (Saturación) son más efectivos y precisos para el objetivo que perseguimos, pues el agente a detectar (la mano), aunque su color es mayoritariamente homogéneo en toda su extensión, las condiciones del ambiente van a influenciar en la percepción del mismo. Un ejemplo, quizás el más destacado, se correspondería con la existencia de sombras que afecten a ciertas regiones de la mano. La conversión de una imagen de un formato a otro se ha realizado a través de la función **cvtColor().**

Realizada la conversión, el siguiente paso consiste en recabar información de la imagen convertida a HLS que nos pueda permitir, en un futuro, discernir qué es una mano de lo que no. Para ello, se va a proceder a determinar una serie de secciones cuadriculares (18 concretamente) de la imagen. En primera instancia, se obtienen las coordenadas que cada una de ellas va a ocupar, y se registra dicha información en un vector de puntos para su uso posterior. Obtenidas las coordenadas, se utilizan para la creación de los susodichos cuadrados a través de la función **rectangle().** Tras este paso, en la imagen aparecerán las secciones marcadas con un color (verde). El usuario deberá cubrir dichas secciones, en su totalidad, con su mano y pulsar una tecla (espacio) para proceder a la captura de un nuevo frame para su análisis. Nuevamente, se convierte la imagen obtenida a HLS y se guarda, en un vector de objetos Scalar, la media del color de cada sección obtenida mediante la función **mean()**.

* 1. **Obtención de la máscara binaria**

El siguiente paso consistirá en obtener la máscara binaria que, en esencia, se corresponderá con una mano. Para ello, se procederá a obtener una serie de imágenes binarias por cada una de las secciones que se crearon en el paso anterior. Estas imágenes son de todo el frame capturado y no de la región contenida en una de las secciones. Para obtener cada una de estas imágenes, se utilizará la función **inRange().** Esta función nos permite obtener una máscara binaria de una imagen al establecer los píxeles de esta a 0 o a 1 en función de si estos están o no dentro de un rango especificado por el programador. Es, en consecuencia, en dichos rangos, donde debemos focalizar nuestra atención.

Los rangos se calculan a través de las medias de color recogidas en el paso anterior. Cada una de estas medias, al ser un objeto Scalar, poseen un indicador de cada canal HLS. Para cada uno de dichos componentes, se fija un valor superior e inferior (por comodidad, se encuentran definidos como atributos). Las correspondientes sumas (con los valores superiores) y restas (con los valores inferiores) de dichos valores con los de cada canal para cada una de las imágenes son almacenados y utilizados para definir, posteriormente, los rangos superiores e inferiores que utilizará la función **inRange()** para crear cada una de las mencionadas imágenes binarias. En caso de que la suma o la resta de los valores supere el valor máximo de un canal o, por el contrario, sea negativo, se ajustará convenientemente a su valor más cercano, es decir, 255 y 0 correspondientemente.

Las imágenes obtenidas mediante **inRange()** son almacenadas en un vector y sumadas sucesivamente a una imagen auxiliar inicialmente en negro. El resultado final de dicho proceso se corresponderá con la máscara binaria obtenida.

* 1. **Ventajas e inconvenientes del método utilizado**

Las ventajas de este método empiezan por la sencillez del mismo, es adaptable al medio, es decir, realmente no necesita que siempre tengamos que ejecutar el programa exactamente en el mismo sitio ya que calculamos las medias al principio de cada ejecución permitiéndonos así la posibilidad de que pueda ser usado por personas distintas e incluso luminosidades distintas.

Por otro lado, este método también tiene varias desventajas como es el hecho de que para calibrar el programa necesitamos que la mano esté a una distancia concreta de la cámara y durante la ejecución del mismo no se aleje demasiado porque dejaría de detectarla. Otra desventaja es el hecho de que para que solo detecte la mano, es necesario que haya un cambio de color de la mano al antebrazo siendo necesario taparlo para no "confundir" al programa ya que se basa en la detección de un rango de colores detectados de las medias recogidas al principio. A pesar de que el programa pueda ser ejecutado bajo distintas limosidades estas pueden afectar si varían durante la ejecución, es decir, el programa podría fallar si durante la ejecución del mismo varía mucho la luz haciendo así que la cámara captase la mano de otro color ligeramente distinto y que se saliese del rango predefinido. De igual manera, si en el entorno de ejecución existiera algún objeto de color similar al de la mano, este podría verse incluido en la máscara binaria al crearse la misma exclusivamente en función de los colores detectados.

1. **Selección del contorno de la mano**
   1. **Reducción del ruido**

La máscara obtenida en el paso anterior presentará, normalmente, diversas cantidades de ruido que deberá ser corregido para obtener una máscara binaria válida que permita la realización con éxito de los pasos posteriores. Para tal fin, OpenCV ofrece distintas funciones que podemos utilizar como distintas alternativas para conseguir nuestro objetivo en función de las situaciones que presente nuestra máscara sin corregir. En caso de que la máscara presente poco ruido, puede ser conveniente realizar operaciones de erosión, que expanden los píxeles negros a sus vecindades. Mientras que si los defectos se encuentran principalmente en el “interior de la mano” puede ser más adecuado emplear primero operaciones de dilatación, que realizan el mismo procedimiento que las de erosión pero extendiendo los pixeles no negros. Ambas funciones se pueden combinar sin temer que una revierta los cambios producidos por la otra, pues las aplicaciones de las funciones se aplican sobre la resultante de la operación anterior, por lo que, por ejemplo, si una operación de erosión se elimina el ruido externo, una posterior operación de dilatación no podrá restaurarlo.

Para nuestro caso concreto, se ha reducido el ruido inicialmente a través de la función **medianBlur()** que realiza un suavizado de la imagen reemplazando cada pixel de la misma por la mediana de sus vecindades localizadas en un cuadrado alrededor del pixel evaluado. Dicho cuadrado, concretamente su tamaño, debe ser especificado por el usuario y, además, impar. Para nuestro caso, se ha optado por un tamaño de 3. La utilización de un filtro basada en el cálculo de una mediana en lugar de una media resulta una opción interesante fundamentalmente porque el color resultante de esta operación es uno que se encuentra en la imagen original, caso distinto al que resultaría de una media.

Posteriormente al filtro de la mediana, se prosiguió con la reducción del ruido a través de una operación de erosión mediante la función **erode(),** con la que se redujo el ruido no perteneciente a la mano.

* 1. **Generación del contorno de la mano**

Para generar el contorno de la mano fue necesario que antes pintáramos un pequeño círculo en la pantalla ya que de lo contrario si el programa no encuentra un contorno fallaría, es decir, siempre debe ser capaz de reconocer un contorno y de esta manera estamos "forzando" el que reconozca al menos el pequeño círculo. En caso de que en el frame aparezca un contorno reconocible mayor el del programa pasa a generar el contorno nuevo en lugar del círculo.

Posteriormente, utilizamos la función **findContours()** para obtener todos los contornos de la imagen y almacenarlos convenientemente en un vector de vectores de puntos, donde cada elemento se corresponde con un contorno distinto y el tamaño del vector interno al vector original, con su extensión; siendo además cada uno de los puntos almacenados las coordenadas por las que transcurre el contorno.

Acto seguido, nos interesa quedarnos únicamente con el contorno que se corresponde con el de la mano. Si los pasos anteriores se han realizado correctamente, el contorno de la mano debería ser el de mayor longitud. En concreto, más que obtener el contorno en sí, nos interesa determinar su índice, es decir, la posición que ocupa dentro del vector de vectores de puntos. Identificado el índice correspondiente, hacemos uso de la función **drawContours()** para dibujarlo en las imágenes captadas por la cámara. La función presentada, en concreto, requiere que se le pasa como argumento la totalidad de los contornos, es decir, el vector de vectores de puntos que sirve como contenedor de estos. Es por ello mismo, por lo que se requiere saber cuál es el índice del contorno que se desea pintar: para indicarle a la función **drawContours()** cuál es el que realmente nos interesa que dibuje sobre la imagen que le indicamos también como parámetro.

1. **Convex Hull**

El siguiente paso por realizar consiste en obtener y pintar sobre la imagen la envolvente convexa de la máscara binaria. Una envolvente convexa (Convex hull en inglés) se define como la intersección de todos los conjuntos convexos que contienen a un conjunto de puntos, o lo que es lo mismo para nuestro caso; un polígono que contiene a toda una serie de puntos en su interior, de área mínima, y donde la intersección entre cualquier de sus puntos se encuentra contenida de igual manera en el polígono.

Para su cálculo, OpenCV nos ofrece la función **convexHull(),** para la cual, es necesario recurrir al contorno de la mano que, previamente, se había identificado del conjunto de contornos detectados. Esto es así ya que dicho contorno será sobre el que se defina la envolvente convexa. No obstante, **convexHull()** únicamente nos permite obtener la información referente a la envolvente convexa, y no representarla de ninguna manera sobre la imagen.

Para lograr su representación, se ha recurrido a un procedimiento “más manual”. Es decir, no se ha usado ningún tipo de función que permita dicha acción en concreto. En su lugar, se ha procedido a utilizar la función **line()** entre los puntos que conformarían la envolvente convexa originando un polígono en el proceso y en el cual se enmarcaría nuestra máscara binaria de la mano. Para tal fin, es necesario recurrir a la información obtenida a través de la función a la que inicialmente hacíamos referencia y que se había obtenido mediante **convexHull().**

1. **Defectos de convexidad**

Una vez se posee tanto el contorno que define la mano, como una envolvente convexa válida, se dispone de todo lo necesario para calcular los defectos de convexidad, los cuales se identifican como los puntos del contorno de la mano más alejados de la envolvente convexa para cada segmento de esta. Para el caso concreto de una mano, estos defectos ayudan a determinar, por ejemplo, el nacimiento de los dedos. Con OpenCV, los defectos pueden ser fácilmente calculados mediante la función **convexityDefects(),** que requiere, como se puede esperar, del contorno de la mano y del hull definido como argumentos. De igual manera, se requiere de algún tipo de contenedor (vector de “enteros de cuatro elementos”) que se pasa por referencia para almacenar los defectos calculados.

No obstante, resulta preciso trabajar con los defectos de convexidad obtenidos con objeto de filtrar aquellos que realmente nos interesan, que en la práctica serán aquellos que definan los extremos de los dedos y las regiones de unión entre ellos. A tal fin, se calcula mediante funciones trigonométricas el ángulo que forman los puntos iniciales, finales y el más lejano de cada defecto encontrado sobre el contorno de la mano y se incluye la condición de que estos han de ser agudos. En consecuencia, si el ángulo calculado entre estos puntos resulta no cumplir con dicha condición, el defecto se descarta. De igual manera, se antepone una condición adicional, que aplica para la distancia que existe entre el punto más lejano del defecto, y la propia envolvente convexa. En concreto, fijamos la condición de que dicha distancia ha de ser superior a un valor que se ajusta en función de la distancia a la que se encuentra la mano de la cámara, hecho que es posible gracias a la implementación de un **boundingRect()** alrededor de la mano.

Adicionalmente, se han representado los defectos de convexidad filtrados sobre la mano mediante el dibujado de círculos de color verde para los defectos en las zonas de unión (puntos finales de los defectos) y de color rojo para los presentes en las puntas de los dedos (puntos iniciales de los defectos).

1. **Contador de dedos**

Una vez implementado todo lo anterior, nos encontramos en una situación apropiada para añadir funcionalidades al programa propias del reconocimiento de gestos. Entre otras, hemos optado por la implementación de un contador de los dedos que la mano tiene levantado en un instante de tiempo determinado.

Esta implementación basa su funcionamiento en la lógica de contar el número de defectos filtrados que presenta la imagen en el frame captado y en función del número de estos, indicar la cantidad de dedos levantados.

El caso base de nuestra implementación se presenta cuando no hay ningún dedo levantado, por lo que se debe indicar por pantalla tal realidad.

En el transcurro de nuestra implementación de la idea propuesta, nos dimos cuenta de una determinada situación: al levantar solo un dedo no se presentaría ningún punto de convexidad verde (consúltese el apartado anterior) por lo que no había ninguna manera de contabilizar que había un dedo. Como solución a este caso, se optó por atender a ambos tipos de puntos. El programa mantendría actualizado en todo momento un contador de ambos, y solo en la situación en la que no se ha registrado ningún punto verde (los cuales se han considerado como más fiables y estables), se atiende a los rojos. Así pues, para el caso que nos atañe, el programa basaría su respuesta en función de estos últimos.

Para el caso general, tal y como se ha indicado, se atiende a los defectos de convexidad representados como puntos verdes. En programa, concretamente, indica que hay una cantidad de dedos levantados igual al número de estos puntos más uno.

1. **Bounding Rect**

Para una mejor implementación del programa y mayor modularidad del mismo hemos optado por usar la función **boundingRect()** para generar el rectángulo mínimo posible del contorno de la mano. Con este rectángulo sacamos proporciones que luego aplicamos a todo el programa.

Un ejemplo de esto es el caso del porcentaje de error que empleamos para la longitud de los dedos. Si pusiéramos un tamaño fijo en píxeles como podría ser por ejemplo 50, esto no sería preciso ya que no equivale a la misma longitud de tu mano 50 píxeles cuando la esta está a 10 centímetros de la pantalla a cuando está a 2 metros.

De esta manera y modulando más el código hemos realizado una ampliación del mismo creando una clase **BoundingRect** que hereda de cv::Rect,cuyos atributos son dos: El punto central de dicho rectángulo y el punto Angular que es el punto de referencia que tenemos para crear diversos triángulos internamente del **BoundingRect** y de está forma calcular rotaciones y demás, aumentando así las funcionalidades que nos pueda brindar: hallar aproximadamente el punto medio de la mano, su inclinación, dirección a la que apuntan los dedos, etc.

1. **Movimiento de la mano**

Esta función se encarga de imprimir en la pantalla hacia donde se está moviendo, se acaba de mover la mano o si esta está parada. Para desarrollar esta función hemos empleado el punto medio del rectángulo generado por el **boundingRect()**.

El funcionamiento de la idea implementada es la de indicar hacia donde se ha movido la mano tras haber realizado un movimiento. Para ello, hemos tenido que basarnos en una idea, consistente esta en comparar la posición actual de la mano en distintos frames para detectar hacia donde se ha movido. El principal problema nos surgió a la hora de determinar cuándo y con qué frames realizar las comparaciones. En principio, surgieron dos ideas distintas sobre cómo afrontar el conflicto. La primera, consistía en contar el número de frames que han pasado, y si estos se corresponden con el número establecido, utilizar el último y primer frames captados para comparar la posición de la mano. El problema de esta opción era que, si utilizábamos una cámara medianamente básica o con un funcionamiento en cuanto a Hz muy básico, la cantidad de frames capturados sería muy baja, por lo que para comparar si se ha movido o no, deberíamos adaptar dicho contador al funcionamiento de la cámara. La segunda idea, que fue por la que optamos finalmente, se basa en utilizar los ciclos de reloj del ordenador, para ello establecemos un contador al igual que para la primera idea, pero la diferencia se basa en que ahora, independientemente de la calidad de la cámara, el registro del movimiento estaría controlado por el dominio del tiempo. De esta manera, cada vez que se cumplen un número exacto de ciclos de reloj, el programa almacena el frame actual, compara el centro de masa de la mano del frame capturado con el de un frame tomado al inicio del proceso de conteo, y en función de hacia donde se haya movido, comparando las coordenadas ‘x’ e ‘y’ del punto, indicaremos por pantalla la posición a la que se ha movido la mano. Una vez realizado todo el cálculo, el programa deberá actualizar el frame de comparación al que hacíamos referencia con el que se captó tras la finalización del contador. De esta manera, garantizamos que en las siguientes iteraciones se compare con un frame representador de la realidad de dicho instante y no del inicio del proceso (entendido como a su vez, el frame inicial captado tras la obtención de la máscara binaria).

1. **Pintar en la pantalla con el dedo**

Por último, se ha desarrollado la funcionalidad de pintar en la pantalla aprovechando los gestos que reconocemos, introducidos ya en el apartado 7.

Centrándonos más en el funcionamiento, tenemos instanciado un vector de pares estático que contendrá los distintos círculos y tamaños que se estarán dibujando por la pantalla. Esto es así con objeto de garantizar que los puntos que se piten en una imagen, se conserven en las venideras. De esta manera, el sistema funciona como una especie de historial al que se recurre con cada frame para saber qué círculos se debe pintar.

Una vez finalizada la estructura que almacenará los puntos, creamos también dos flags, necesarios para activar la función de pintado. (**ChangePaintState\_** y **paint\_**). La primera variable dicta si es posible cambiar al estado de poder pintar y la segunda, si se está pintando. En consecuencia, será la alteración de estos flags los que regulen el proceso de gestión de pintado sobre los frames que capten la cámara. El modo de pintado solo puede ser activado cuando lo permite el flag correspondiente, es decir, el de cambio. Este flag cambia su valor cuando, partiendo de la situación inicial de la mano cerrada (en puño), se muestra cualquier otra cantidad de dedos. El segundo flag, el que regular el modo de pintado en sí mismo, únicamente puede activarse o desactivarse al cerrar la mano siempre y cuando el otro flag lo permita. El motivo de esta implementación en sencillo, y consiste en evitar una situación en la que el gesto que activa y desactiva el modo de pintado cambio el valor del flag correspondiente constantemente debido a la captación de múltiples frames con el mismo gesto en un corto instante de tiempo.

Una cuestión de importancia, consiste en indicar que los círculos que se dibujan en las imágenes captadas pueden presentar distintos radios. Esto es debido a que este parámetro no es fijo, y se ha adaptado para que varíe en función de la distancia de la mano a la cámara. En concreto, cuanta más cerca esté la mano de la cámara, más grande se verá el círculo, hecho que es posible gracias a una relación establecida con el boundingRect previamente establecido.

Cómo aclaración final, destacar que si se desea borrar el pintado sobre las imágenes, basta con desactivar el modo de pintado. De igual manera, pintar será solo posible cuando únicamente se presente un dedo levantado ante la cámara, y será la punta de este la que actúe como pincel. Si el modo de pintado está activado y se exponen más dedos, los círculos hasta dicho entonces dibujados se mantendrán, pero no se crearán nuevas hasta que se vuelva a dar la condición previamente descrita o, por el contrario, se desactivo el modo de pintado.