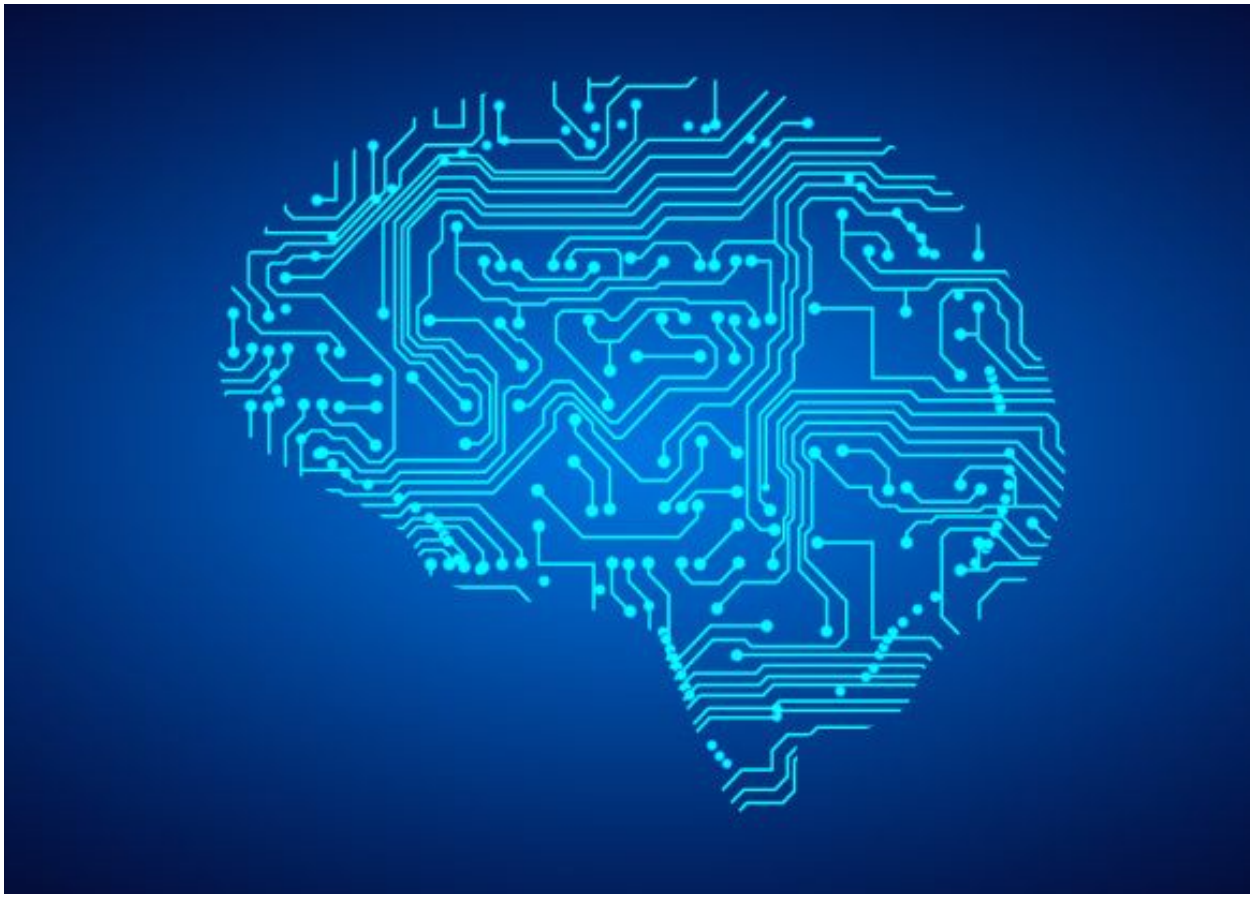
Práctica SIPC

Reconocimiento de Gestos



Autores

Ángel Luis Igareta Herráiz
Cristian Manuel Abrante Dorta
Carlos Domínguez García

Introducción

En esta práctica hemos construido un sistema que permita el reconocimiento de gestos con las manos de forma automática. Para ello hemos necesitado familiarizarnos con los distintos procesos involucrados para llevar a cabo una técnica de reconocimiento de gestos: sustracción de fondo además de detección del contorno y características de la mano.

Para el desarrollo de la misma hemos utilizado la librería OpenCV. Se trata una librería de visión artificial y aprendizaje automático, diseñada para acelerar la implantación de sistemas automáticos de percepción. Dispone de 2500 algoritmos.

Características de nuestro proyecto

A continuación se mencionan las características de nuestro proyecto:

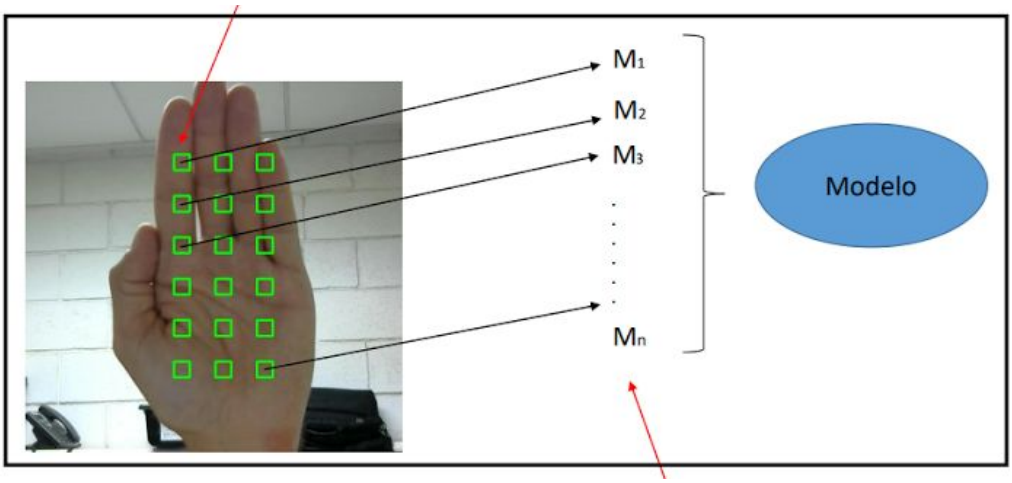
- Reconocimiento de dedos de la mano levantados con el fin de **mostrar un contador**. Cabe notar que como nos hemos centrado en las siguientes características, ésta la hemos dejado en una fase bastante básica, por ejemplo no reconoce el cero.
- Seguimiento de la mano para **pintar en un lienzo virtual**.
- Seguimiento del área de la mano para **cambiar de color** cuando haya un cambio brusco.
- Detección de un patrón de movimiento al dibujar (derecha-izquierda-derecha) para **dejar de pintar**.
- Detección de un número específico de dedos para empezar y terminar de pintar, así como para **borrar el lienzo**.

Procesos

1. Sustracción del fondo

La sustracción del fondo consiste en eliminar el fondo de una imagen para obtener la región de interés que corresponda a la mano. Como trabajamos en luz visible tenemos que tener en cuenta el rango de color que tiene una mano.

Para obtener el rango de color se muestra una primera ventana llamada **"Reconocimiento"** en la que se dibujan un total de 18 cuadrados verdes, que serán nuestras regiones de interés. En ellos deberemos colocar la mano de la que queremos que se halle el modelo y pulsar *espacio*.




```
// Store the borders of the squares that will be painted.
for( int i = 0; i < MAX_HORIZ_SAMPLES; i++ ) {
   for( int j = 0; j < MAX_VERT_SAMPLES; j++ ) {
       cv::Point p;
       p.x = frame.cols / 2 + ( -MAX_HORIZ_SAMPLES / 2 + i ) * (
SAMPLE_SIZE + DISTANCE_BETWEEN_SAMPLES );
       p.y = frame.rows / 2 + ( -MAX_VERT_SAMPLES / 2 + j ) * ( SAMPLE_SIZE
+ DISTANCE_BETWEEN_SAMPLES );
       samples_positions.push_back( p );
    }
}
```

A continuación se hallará la media de los colores de cada una de las regiones de interés utilizando **cv::mean** y se guardará en un vector de **cv::Scalar**, utilizados para almacenar valores de píxeles.

Cabe añadir que previamente deberemos pasar el modelo de color de la imagen a **HLS**. El modelo HSL o HSI significa **Hue, Saturation, Lightness** y se utiliza dado que para hacer medias y saber los colores parecidos es mucho mejor que RGB, pues podemos variar el matiz, saturación y luminosidad y obtener parecidos.

```
// Convert the actual image to HSL Color Model.
cv::cvtColor( frame, hsl_frame, CV_BGR2HLS );

// Get Region Of Interest and calculate the mean.
for( int i = 0; i < max_samples; i++ ) {
    cv::Mat roi = hsl_frame( cv::Rect( samples_positions[i].x,
samples_positions[i].y, SAMPLE_SIZE, SAMPLE_SIZE ) );
    means[i] = cv::mean( roi ); // Calculate the pixel's average.
}
```

A continuación por cada media deberemos obtener el rango de color de cada canal HSL. Para ello primero deberemos convertir la imagen al modelo HSL y generar una imagen binaria con canal de 8 bits que actuará como acumulador.

```
// CODE 1.2: For each mean, get the range of each HSL channel.
cv::Mat hsl_frame;
cv::cvtColor( frame, hsl_frame, CV_BGR2HLS );

// Instance an Image ACC with 8-bit single channel image.
cv::Mat acc( frame.rows, frame.cols, CV_8UC1, cv::Scalar( 0 ) );
```

Tras esto, por cada media deberemos obtener el rango desde 0 a 255 con **cv::inRange**.

```
cv::inRange( hsl_frame, lower_bounds[i], upper_bounds[i], bgmask );
```

Lo que devuelve esta función es un array con los píxeles que cumplan la siguiente condición:

$$\mathtt{dst}(\mathrm{I}) = \mathtt{lowerb}(\mathrm{I})_0 \leq \mathtt{src}(\mathrm{I})_0 \leq \mathtt{upperb}(\mathrm{I})_0$$

Lo único que quedaría sería ir sumandolo al acumulador.

```
acc += bgmask;
```

1.1 Ventajas e Inconvenientes

Gracias a utilizar el modelo de color HLS hemos pedido utilizar un trackbar para modificar los valores altos y bajos del matiz, luminosidad y saturación a tiempo real. De esta manera hemos podido obtener una imagen retocada a nuestro gusto que la podamos adaptar al ambiente en el que estemos.

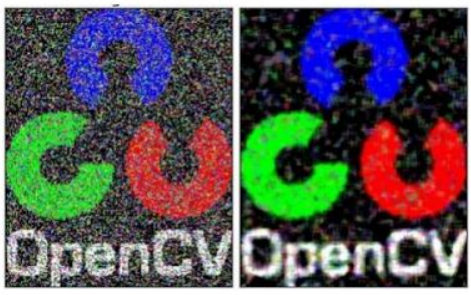
Por otra parte si no se toman bien las regiones de interés en la etapa de reconocimiento se generaría una media de color distinta a las demás y también se sumaría al acumulador. Esto provocaría que los colores del fondo también se tomaran como los de la mano y, por lo tanto, no sería capaz de reconocer nuestra mano, ya que la variedad de colores sería alta.

2. Reducción del ruido

Un problema al que debemos enfrentarnos es al ruido que genere la propia cámara pues el ambiente puede ser muy desfavorable, es decir poco iluminado. Para solventar de la mejor forma posible este problema utilizaremos dos operaciones.

2.1 Filtro de la mediana

La idea principal del filtro de la mediana es ir pixel por pixel reemplazandolo con la **mediana** de los pixeles vecinos, conocidos como “ventana”. Para hacer ésto primero se ordenan numéricamente y luego se coge el pixel que queda en medio.



Ventajas frente a la media:

- La mediana es mucho más robusta, pues la media puede coger un valor muy separado a los demás y afectar mucho en el resultado. Sin embargo en la mediana no afectaría tan significativamente.
- A su vez, tampoco crea un valor que **no existe**, si no cogería uno de la ventana.

Para utilizarla en nuestro programa usamos **cv::medianBlur**, cuyos parámetros son:

```
void cv::medianBlur(InputArray src, OutputArray dst, int ksize)
```

Donde **ksize** es el tamaño de la ventana. En nuestro programa utilizamos una venta de tamaño 5.

```
// CODE 2.1: Reduce mask noise.
cv::medianBlur( bgmask, bgmask, 5 );
```

2.2 Operaciones morfológicas

- **Dilatación:** La operación de dilatación consiste en **"retorcer"** una imagen A con alguna forma, usualmente un cuadrado o círculo. De esta manera las regiones más blancas (distintas de 0) tienden a **"crecer"** (de ahí dilatación). Una ventaja de utilizarla es cuando en el centro de nuestro modelo existen muchos huecos negros, pues gracias a la dilatación desaparecerían.

- **Erosión:** La operación de erosión es la hermana de la dilatación y lo que hace es expandir los píxeles negros (iguales a 0). Por lo tanto, una ventaja de utilizarla es cuando en en el fondo de nuestro modelo existen muchos puntos blancos, pues dado que expandimos los píxeles negros desaparecerían.

En nuestro programa utilizamos la dilatación con la elipse como forma morfológica. De tamaño de dilatación elegimos dos.

```
int dilation_size = 2;
cv::Mat element = cv::getStructuringElement( cv::MORPH_ELLIPSE,
        cv::Size( 2 * dilation_size + 1, 2 * dilation_size + 1 ),
        cv::Point( dilation_size, dilation_size ) );

cv::dilate( bgmask, bgmask, element );
```

Como hemos visto las operaciones morfológicas y el filtro de la mediana son herramientas muy potentes para la reducción del ruido. Podemos elegir si usamos los dos o una de ellas.

También es importante el orden en el que usamos las operaciones morfológicas. Dependiendo de nuestro modelo, hay que tener cuidado dado que los pixeles que desaparezcan en una de las operaciones no se puede recuperar por mucho realizar la operación inversa.

3. Operaciones con la imagen binaria

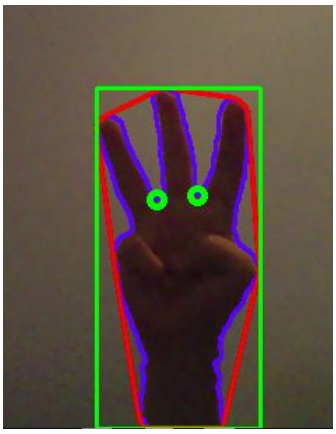
Una vez tengamos una imagen binaria de la mano, es decir que tengamos separados los píxeles de interés, realizaremos operaciones con la misma. Cabe destacar que en la imagen binaria los píxeles de interés estarán pintados en blanco (valor 1) y los píxeles del fondo estarán pintados en negro (valor 0).

La operaciones sobre la imagen binaria las realizaremos en el método `feauresDetection` de la clase `HandGesture`.

```
std::vector<cv::Point> FeaturesDetection(cv::Mat mask, cv::Mat output_img,
double &countour_area);
```

Este método recibe los siguientes parámetros:

- **`cv::Mat mask`:** Es la máscara binaria de la imagen con el fondo separado.
- **`cv::Mat output_img`:** Es la imagen que se devolverá. Será la imagen captada por la cámara pero con el contorno de la mano, la malla convexa, el rectángulo mínimo y los diferentes puntos medios de los dedos pintados. Será similar a esta:



- **`double &contour_area`:** es el área del rectángulo que engloba a los puntos del contorno.

Además devuelve un vector de puntos (`std::vector<cv::Point>`) que se corresponden con cada uno de los puntos medios de los dedos que en ese momento estén levantados. Explicaremos por separado las operaciones que hemos llevado a cabo:

3.1 Detección del contorno de la mano

Lo primero que haremos será detectar el contorno de la mano en la imagen binaria. Con contorno nos referimos a la línea exterior de una zona cerrada pintada de color blanco. Para ello utilizaremos la función `cv::findContours` de esta forma:

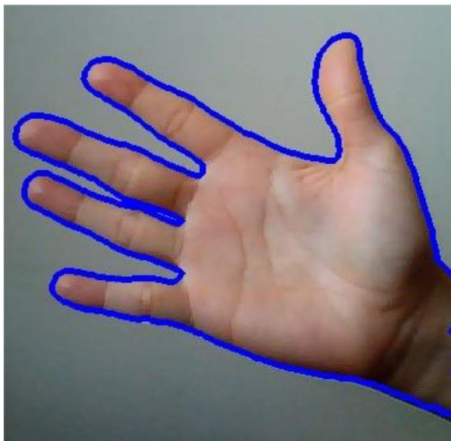
```
// Vector de vectores de puntos utilizado para almacenar el contorno.
std::vector<std::vector<cv::Point> > contours;

// Copiamos la imagen original a una temporal.
cv::Mat temp_mask;
mask.copyTo( temp_mask );
```

```
// Encontramos los contornos.
findContours(temp_mask, contours, CV_RETR_EXTERNAL, CV_CHAIN_APPROX_SIMPLE);

// Buscar el contorno más largo.
int max_contour_index = 0;
for( int i = 1; i < contours.size( ); ++i ) {
    if( contours[max_contour_index].size( ) < contours[i].size( ) ) {
       max_contour_index = i;
    }
}

// Pintar el contorno en la imagen de salida.
drawContours(output_img, contours, max_contour_index,
             cv::Scalar( 255, 0, 0 ), 2, 8,
             std::vector<cv::Vec4i>( ), 0, cv::Point( ) );
```

Los contornos se almacenarán en un vector de vectores de puntos, porque puede que en la imagen hayan varios contornos. Por ello, lo que haremos será buscar el contorno más grande, ya que este probablemente corresponderá al contorno del área de la mano.

Cabe destacar que durante la búsqueda del contorno, la función `cv::findContours` modifica la imagen que le pasemos como parámetro, por ello es necesario que hagamos una copia y la pasemos a una imagen temporal (`mask.copyTo(temp_mask)`).

Tras haber obtenido el índice del contorno máximo (`max_contour_index`) lo que haremos será pintarlo en la imagen de salida mediante la función `cv::drawContours`. Por tanto la imagen de salida (`output_img`) será igual a la imagen original, sin embargo tendrá añadido el dibujo del contorno de la mano en color azul.



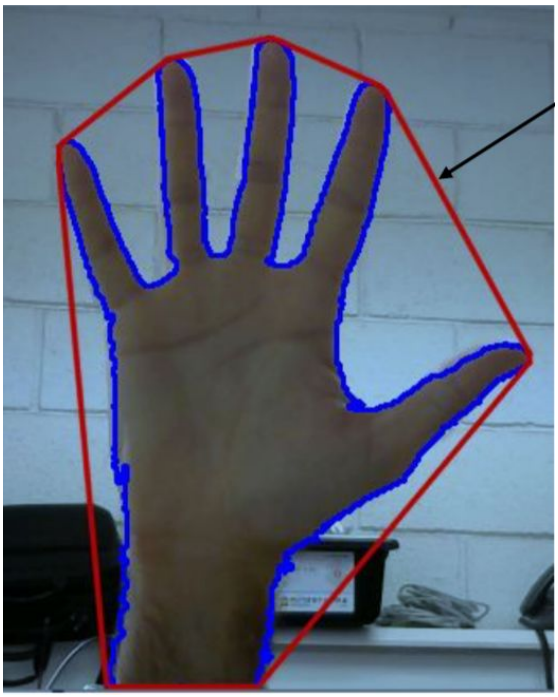
3.2 Obtenemos la malla convexa (convex hull)

La malla convexa o convex hull se corresponde con el polígono convexo que contiene a un conjunto de puntos dado, y donde sus vértices son alguno de estos puntos. Nuestra intención es calcular la componente conexa del contorno de la mano (que es un conjunto de puntos). De esta forma podremos obtener un polígono cuyos vértices serán las puntas de los dedos. Para hacer esto utilizamos la función `cv::convexHull` de esta forma:

```
std::vector<int> hull;
convexHull( contours[max_contour_index], hull );

// pintar el convex hull
cv::Point pt0 = contours[max_contour_index][hull[hull.size( ) - 1]];
for( int i = 0; i < hull.size( ); i++ ) {
    cv::Point pt = contours[max_contour_index][hull[i]];
    line( output_img, pt0, pt, cv::Scalar( 0, 0, 255 ), 2, CV_AA );
    pt0 = pt;
}
```

La malla convexa se almacenará como un vector de índices (`hull`), es decir los puntos que se corresponderán con los vértices del polígono convexo. Tras calcular dicho vector deberemos pintar los puntos. Para ello utilizamos la función `cv::line` que dibuja un segmento entre los puntos que indiquemos. De esta forma lo que hacemos es recorrer el vector de índices, pintando un segmento por cada pareja de puntos.

Este polígono convexo lo pintaremos también en la imagen de salida (`output_img`) en color rojo.

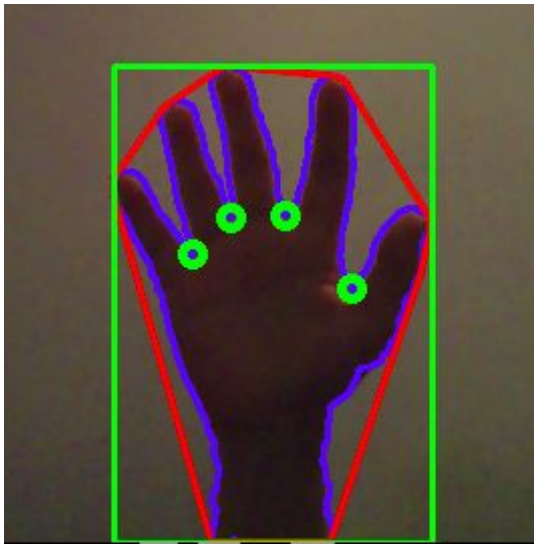


3.2 Obtenemos el rectángulo que contiene los puntos (bounding rect).

A continuación lo que haremos será determinar el rectángulo mínimo que contiene a los puntos del contorno. De esta forma podremos determinar el área aproximada de la mano. Para determinar el rectángulo de área mínima utilizamos la función `cv::boundingRect`:

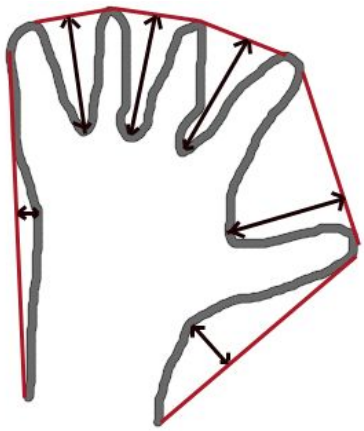
```
cv::Rect bounding_rect = cv::boundingRect(contours[max_contour_index]);
cv::rectangle( output_img, bounding_rect.tl(), bounding_rect.br(),
cv::Scalar( 0, 255, 0 ), 2, 8, 0);
countour_area = bounding_rect.area();
```

En primer lugar creamos un rectángulo (`cv::Rect bounding_rect`) en el cual almacenamos el resultado de `cv::boundingRect` con contorno máximo. A continuación, pintamos en color verde el contorno de dicho rectángulo en la imagen de salida mediante la función `cv::rectangle`. Finalmente al parámetro del contorno del área (`contour_area`) le damos el valor del área que tiene el rectángulo que hemos creado. De esta forma podremos obtener el área del rectángulo mínimo que engloba a la mano.



3.3 Obtenemos los defectos de convexidad

Los defectos de convexidad son zonas que difieren de la malla convexa de la imagen. Estas zonas realmente no pertenecen a la mano que estamos intentando reconocer, aunque si que son útiles para llevar a cabo ciertos cálculos sobre ella. Los defectos de convexidad pueden representarse por su punto de inicio, su punto final y el punto donde la convexidad es máxima. Este esquema ilustra muy bien el concepto:



Para hallar los defectos de convexidad de la imagen utilizaremos la función `cv::convexityDefects`:

```
//obtener los defectos de convexidad
std::vector<cv::Vec4i> defects;
```

```
convexityDefects( contours[max_contour_index], hull, defects );

// Dibujar los puntos de la mano
for( int i = 0; i < defects.size( ); i++ ) {
  // Punto de start (mas cercano atras en la malla convexa)
  cv::Point start_point = contours[max_contour_index][defects[i][0]];
  // Punto de end (mas cercano delante en la malla convexa)
  cv::Point end_point = contours[max_contour_index][defects[i][1]];
  // Punto furthest (Punto mas separado de los dos anteriores, formando un ángulo)
  cv::Point furthest_point = contours[max_contour_index][defects[i][2]];

  cv::Point middle_point;
  middle_point.x = ( start_point.x + end_point.x ) / 2;
  middle_point.y = ( start_point.y + end_point.y ) / 2;

  float depth = (float) defects[i][3] / 256.0;
  double angle = getAngle( start_point, end_point, furthest_point );

  // Nuestros dedos sólo pueden llegar a 90º, asi que no mostramos el resto.
  if( angle < 90 ) {
     circle( output_img, furthest_point, 5, cv::Scalar( 0, 255, 0 ), 3 );
     resulting_points.push_back( middle_point );
  }
}

return resulting_points;
```

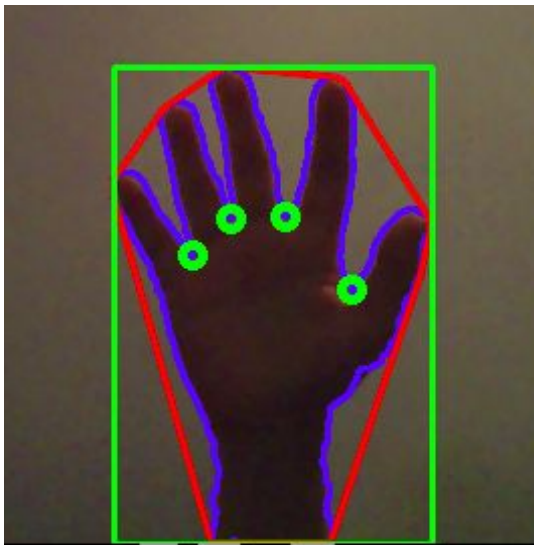
Los defectos de convexidad se almacenan en un vector de vectores con 4 componentes (`std::vector<cv::Vec4i> defects`). La primera componente es el punto inicial de la zona convexa (`start_point`), la segunda componente es el punto final (`end_point`), la tercera es el punto más alejado o máximo de la convexidad (`furdthest_point`) y la cuarta es la distancia entre el punto máximo y la malla convexa (`depth`).

Por ello, lo que hacemos inicialmente es calcular los defectos del contorno máximo (`contours[max_contour_index]`) con la malla convexa que hemos obtenido previamente (`hull`). Seguidamente recorremos el vector de defectos mediante un bucle for.

Dentro del bucle asignamos a variables las componentes del vector de defectos además de calcular el punto medio (`middle_point`). Este punto medio es el resultado de la intersección perpendicular entre la recta que proyecta el punto máximo de convexidad y la malla convexa, o lo que es lo mismo, la mitad entre el punto de inicio y final. Además, lo introduciremos en un vector (`resulting_points`) que retornaremos para utilizarlo posteriormente.

Luego calculamos el ángulo que forman entre sí los segmentos que se producen de la unión entre el punto más alejado y el inicio y fin de la zona de convexidad. Hemos determinado que los dedos de la mano pueden tener entre sí como máximo 90º por tanto si el ángulo es menor que esa cantidad podemos afirmar que el defecto de convexidad corresponde con un dedo. En ese caso pintamos un círculo verde en el punto más alejado mediante la función `cv::circle` e introducimos el punto medio en el vector resultante.

Mediante este método podremos obtener un punto por cada pareja de dedos que esté levantada porque solo una pareja de dedos producirá un defecto de convexidad.



4. Aplicaciones implementadas

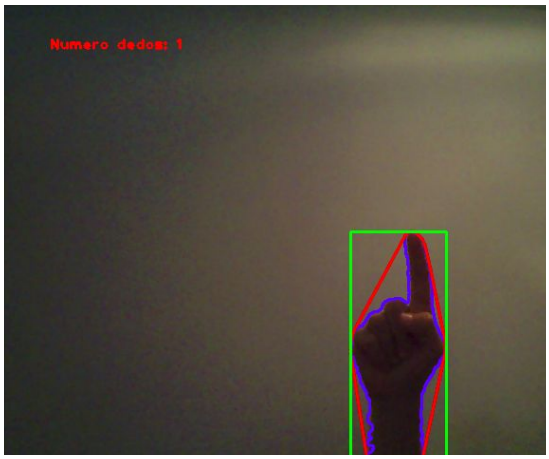
Una vez ya hemos procesado la información procedente de la máscara y hemos determinado todas sus características implementaremos una serie de aplicaciones:

4.1 Contador de dedos

La aplicación más sencilla que podemos realizar con nuestro programa es un contador de los dedos. Para realizarlo, lo que hacemos es mirar el tamaño del vector de puntos medios devuelto por la función `FeautresDetection`. El número de dedos que tendremos levantados será uno más que el tamaño de dicho vector:

```
finger_number = resulting_points.size( );
cv::putText( frame, "Numero dedos: " + std::to_string( finger_number + 1 ),
cv::Point( 50, 50 ), cv::FONT_HERSHEY_PLAIN, 1.0, CV_RGB( 255, 0, 0 ), 2.0
);
```

Utilizamos la función `cv::putText` para imprimir el número de dedos en color rojo en la imagen captada por la cámara.



Un caso raro en este método es que no es capaz de distinguir si tenemos un dedo levantado o ninguno, tal y como podemos ver:

4.3 Pintar con gestos y detección de movimientos

Nuestro programa es capaz de pintar una ventana con el movimiento que se está generando en nuestra mano. Además, también es capaz de detectar la dirección de dicho movimiento.

Antes de comenzar inicializamos las siguientes variables:

```
// Variables utilizadas para pintar
int finger_number = -1;
bool draw_mode = false;
auto random_color = cv::Scalar( 72, 98, 235 );
double previous_area = 0.0;
cv::Point previous_point = {-1, -1};

// Para detectar el movimiento del usuario al pintar
std::vector<int> movements = {0, 0, 0}, expectedMoves = {-1, 1, -1};
int moveIndex = 0, currentMove = 0;
```

Que almacenarán estos valores:

- **finger_number:** número de dedos que el usuario tiene levantados.
- **draw_mode:** variable que determina si el usuario está en el modo dibujo o no.
- **random_color**: color aleatorio con el que se comenzará a pintar la pantalla, si no se quiere cambiar utilizamos el color `#ec6348`.
- **previous_area**: Área anterior que se utilizará para determinar el tamaño de la mano.
- **movements:** vector que almacenará los tres últimos movimientos. Los movimientos se codificarán como: derecha (-1), parado (0) y izquierda (1).
- **expectedMoves:** vector de movimientos esperados para dejar de pintar. Se corresponde con un movimiento: derecha, izquierda, derecha.
- **moveIndex:** índice dentro del vector de movimientos. Este se tratará como un vector circular.
- **currentMove:** movimiento actual. Puede ser izquierda, parado o derecha.

A la hora de empezar a pintar ejecutamos el siguiente código:

```
if( finger_number != resulting_points.size( ) ) {
  finger_number = resulting_points.size( );

  // Empezamos o Acabamos el modo dibujo si tiene los 5 dedos
  // O si el usuario ha pintado siguiendo el patrón de
  // movimientos derecha-izquierda-derecha
  if( finger_number == 4 || movements == expectedMoves) {
    movements = {0, 0, 0};
    moveIndex = 0;
    currentMove = 0;

    // Si ya estabamos jugando acabamos
    if( draw_mode ) {
      draw_mode = false;
      draw.setTo( cv::Scalar( 255, 255, 255 ) );
      previous_point = {-1, -1};
      cv::putText( frame, "Juego finalizado. Cierre y abra para jugar de
nuevo.", cv::Point( 50, 50 ), cv::FONT_HERSHEY_PLAIN, 1.0, CV_RGB( 255, 0,
0 ), 2.0 );
    }

    // Si no estabamos jugando
    else {
      draw_mode = true;
      frame.copyTo( draw );
      draw.setTo( cv::Scalar( 255, 255, 255 ) );
      previous_point = {-1, -1};
     }
  }
}
```

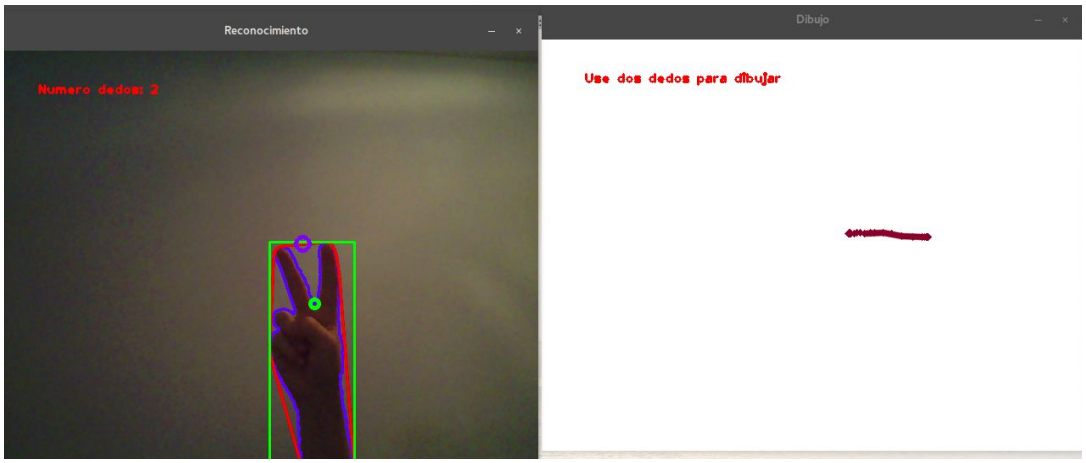
Si el número de dedos ha cambiado, procesamos la información. Si el usuario tiene 5 dedos levantados, entonces reseteamos el vector de movimientos y comprobamos si ya estaba pintando. Si estaba pintando entonces finalizamos el juego. En el caso de que no estuviera pintando, comenzamos este modo y creamos una ventana completamente blanca en la cual pintaremos (`draw`).

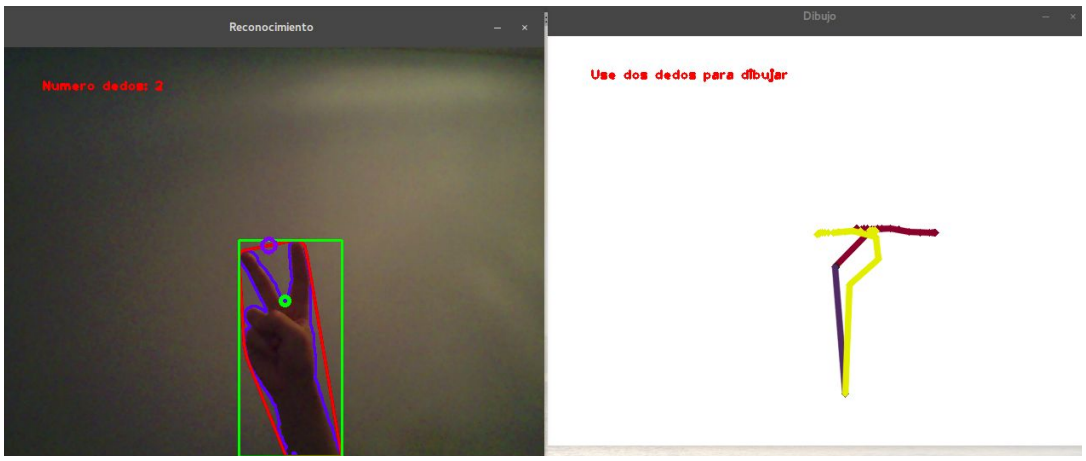
Cabe destacar que para comenzar este modo, el usuario ha de tener levantado los 5 dedos.

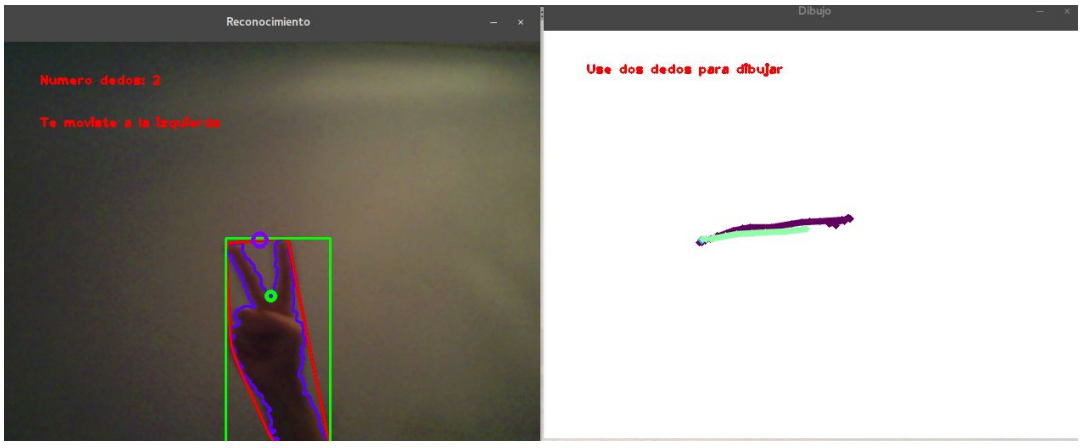
Seguidamente pasamos a la fase en la que se pinta en la pantalla y a la vez se detecta el movimiento de la mano:

```
if( finger_number == 1 ) {
  cv::circle( frame, resulting_points[0], 8, cv::Scalar( 255, 60, 100 ), 3 );
  cv::circle( draw, resulting_points[0], 2, random_color, 3 );
  if( previous_point == cv::Point(-1, -1) ) {
    previous_point = resulting_points[0];
  }
  auto result = resulting_points[0] - previous_point;
  if( cv::norm( result ) > 20 ) { // Euclidian distance

    std::string message = "";

    // Hallamos el movimiento del usuario
    // Si hay un cambio de movimiento lo guardamos en el vector
    if (result.x < 0){
      message += "Te moviste a la derecha ";
      if (currentMove != -1) {
        currentMove = -1;
        movements[moveIndex%movements.size()] = currentMove;
        moveIndex += 1;
      }
     } else if(result.x > 0){
       message += "Te moviste a la izquierda ";
       if( currentMove != 1 ){
         currentMove = 1;
         movements[moveIndex%movements.size()] = currentMove;
         moveIndex += 1;
       }
     }

     cv::putText( frame, message, cv::Point( 50, 100 ), cv::FONT_HERSHEY_PLAIN,
1.0, CV_RGB( 255, 0, 0 ), 2.0 );

     cv::line( draw, resulting_points[0], previous_point, random_color, 5, CV_AA
);
}
```

El primer caso es que el número de dedos sea 1. En esa situación pintamos un círculo en la pantalla de dibujo en la misma posición del punto medio de los dos dedos. Luego calculamos la distancia entre el punto actual y el anterior. Si esta distancia es mayor que 20 determinamos la dirección del movimiento según el signo de la componente x de las resta entre ambos puntos.







FInalmente, dibujamos un mensaje con el movimiento y una línea entre el punto anterior y el actual, con el color que tengamos en el momento.

El segundo caso es cuando estamos pintando pero el área de la mano ha cambiado (cerramos el puño):

```
// Cambiar de color si hay un cambio brusco del área de la mano
// o si el usuario movio la mano derecha-izquierda-derecha
else if( std::abs( countour_area - previous_area ) < ( 0.01 *
std::min(countour_area, previous_area) ) ) {
  int random_red = rand( ) % 256;
  int random_green = rand( ) % 256;
  int random_blue = rand( ) % 256;
  random_color = cv::Scalar( random_blue, random_green, random_red );
}
// Mostrar al usuario ayuda
else {
  cv::putText( draw, "Use dos dedos para dibujar", cv::Point( 50, 50 ),
cv::FONT_HERSHEY_PLAIN, 1.0, CV_RGB( 255, 0, 0 ), 2.0 );
}
```

En el caso de que el área cambien en más de un 10% del área anterior entonces querrá decir que el usuario quiere cambiar de color. Por tanto elegiremos un color aleatorio.

Conclusiones

La librería openCV tiene una gran cantidad de algoritmos y posibilidades en cuanto a procesamiento de imagen se refiere. La práctica que hemos realizado es interesante porque nos brinda una visión general de esta librería, permitiéndonos conocer muchas de sus funciones más importantes.