**Capítulo 1**

**Desarrollo**

* 1. **Desarrollo de detección y reconocimiento de video**

Para este proyecto, ha sido necesaria la implementación de técnicas de procesamiento de imagen, con el objetivo de crear funciones de reconocimiento de imágenes para que el drone correspondiente, pueda actuar en consecuencia a algo que haya alrededor y él haya reconocido.

En este apartado, vamos a expresar primeramente las funciones de preprocesamiento de imagen que hemos desarrollado en C#. Después detallaremos un poco más a fondo la detección y el reconocimiento que se está realizando actualmente. Por último dedicaremos un apartado a otros algoritmos de reconocimiento estudiados.

Todo el desarrollo de imágenes está basado en la librería para C# de EmguCv, que es una envoltura o “*wrapper*” del ya conocido OpenCv. Esto se decidió así, debido a la gran cantidad de funciones y procedimientos que ya están implementados en dicha librería. Además al ser una librería abierta, permite su actualización continua, lo que es un gran apoyo a nuestra API.

* + 1. **Preprocesamiento**

El preprocesamiento, en nuestro caso, es todo el procesamiento de imágenes que puede recibir una imagen antes de pasar a la etapa de reconocimiento y detección.

Para ello se han generado varias funciones que pueden ser útiles para cualquier persona que desee desarrollar software con nuestra API.

* *Filtro de color*

Este filtro se encarga de, según el color seleccionado, ir píxel a píxel de la imagen pintando del color seleccionado aquellos que coincidan en un rango definido con dicho color.

Para ello se ha realizado un desarrollo software importante, debido a que este filtro no lo encontramos en EmguCv. La función desarrollada recibe una imagen como entrada, lee valores del rango aceptado y del color seleccionado desde unas variables globales, y devuelve la imagen correspondiente una vez pasada por el filtro.

A continuación podemos ver de un modo simple, cual ha sido el desarrollo:

/\* VARIABLES GLOBALES \*/

public static Double bluemin = 0;

public static Double bluemax = 0;

public static Double greenmin = 0;

public static Double greenmax = 0;

public static Double redmin = 0;

public static Double redmax = 0;

public static Bgr colorSelected = new Bgr(0, 0, 0);

/\* FUNCION FILTRO DE COLOR \*/

public static Image<Bgr, Byte> filterColor(Image<Bgr, Byte> image){…}

/\* FUNCIONES AUXILIARES \*/

public static Boolean isNotInRange(Bgr now\_color, Double redmin, Double redmax, Double bluemin, Double bluemax, Double greenmin, Double greenmax){…}

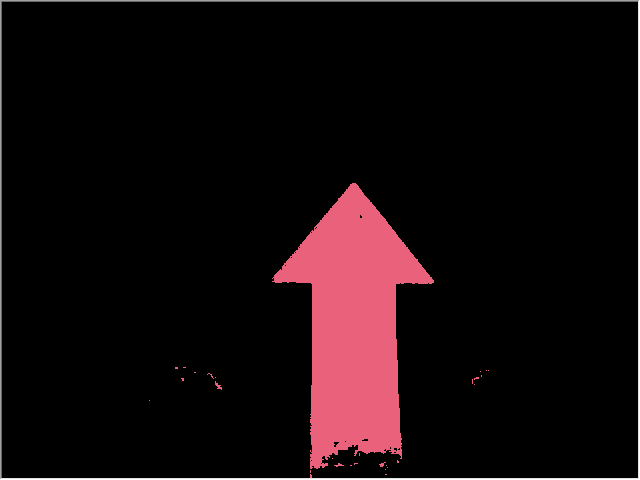
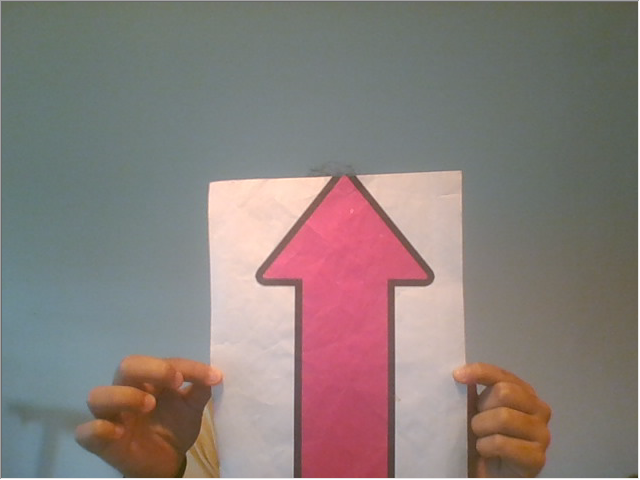
La función auxiliar comprueba si el color pasado por parámetro está dentro del rango o no, del color seleccionado en el filtro.

Las variables globales, sirven para definir el rango y el color seleccionado.

La función filtro de color, recorrerá la imagen píxel a píxel, comprobará si el color está en el rango o no. Si es un sí lo pintará del color seleccionado, si por el contrario es un no, lo pintará de negro.

En conclusión podemos ver el resultado con esta comparativa aplicada al procesamiento en vídeo:

Figura 1. Imagen original a la izquierda. Imagen con el filtro de color a la derecha.



Como vemos el resultado es bastante bueno. Esto implica que el filtro nos puede servir para muchas aplicaciones.

* *Filtro de eliminación de ruido de sal y pimienta*

Otro filtro, ya bastante conocido, es el de eliminación del ruido sal y pimienta. El ruido es lo que se conoce como la variación aleatoria del brillo o el color en las imágenes producido por el dispositivo que recoge o crea la imagen.

El ruido de sal y pimienta se conoce por este nombre porque consiste en píxeles blancos (sal) y negros (pimienta) que se generan por toda la imagen de forma aleatoria. Un ejemplo de imagen con ruido de sal y pimienta puede ser este:

Figura 2. Imagen con ruido de sal y pimienta.

Como vemos la imagen tiene mucho ruido de sal y pimienta. Para eliminar este ruido se utiliza el filtro de la mediana que ya está implementado en EmguCv, con lo que su desarrollo ha sido sencillo.

De todas formas para comprender su funcionamiento explicaremos con un poco de detalle lo que realiza.

El filtro de sal y pimienta consiste puramente en realizar la media de valores de los píxeles circundantes, según una cuadrilla (en nuestro caso 3x3), a cada píxel y sustituir dicho píxel por dicho valor, lo que elimina todos los puntos de sal y pimienta de la imagen haciendolos inapreciables.

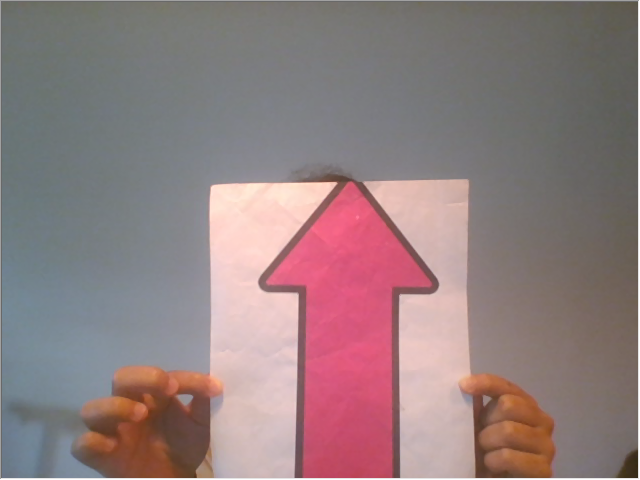
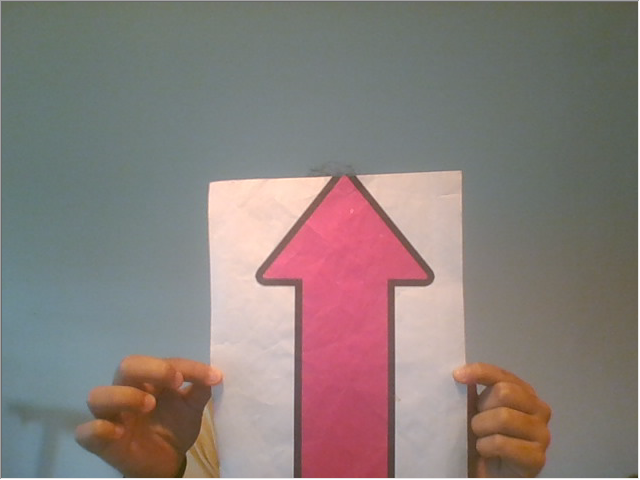
El desarrollo realizado ha sido sencillo ya que está función estaba implementada en EmguCv. La función recibe una imagen y devuelve otra resultante de aplicar el filtro.

/\* FUNCION FILTRO DE ELIMINACION DE SAL Y PIMIENTA \*/

public static Image<Bgr, Byte> filterRemoveSaltAndPepper(Image<Bgr, Byte> image){…}

No se va a apreciar muy bien en vídeo pues nuestra cámara no otorga dicho ruido, pero aún así incluimos una comparativa de este filtro:

Figura 3. Imagen original a la izquierda. Imagen con el filtro de sal y pimienta a la derecha.



* *Filtro de eliminación de ruido gaussiano*

El ruido gaussiano produce en la imagen diversas variaciones de color, este filtro usará la función de Gauss para suavizar los píxeles de una forma similar a la que ya vimos con el filtro de la mediana.

El efecto es un suavizado de la imagen bastante mejorado, con los bordes más denotados. Debido a esto, este tipo de filtros también puede servir para eliminar el ruido de sal y pimienta.

La función de Gauss, en una dimensión, que es nuestro caso, se calcula a partir de la siguiente función:



La frecuencia viene dada por la transformada de Fourier:



Por último estas dos fórmulas se pueden expresar como una desviación estándar:

Utilizando por tanto dicha fórmula, se aplicará a cada píxel, suavizando la imagen.

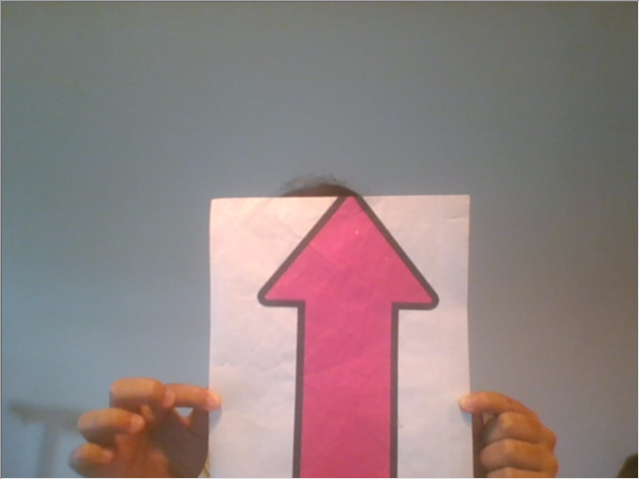
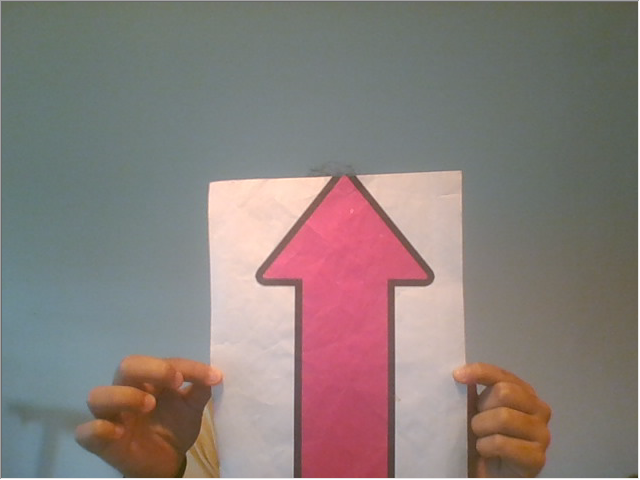
El desarrollo realizado ha sido sencillo ya que está función estaba implementada en EmguCv. La función recibe una imagen y devuelve otra resultante de aplicar el filtro.

/\* FUNCION FILTRO DE ELIMINACION DE RUIDO DE GAUSS \*/

public static Image<Bgr, Byte> filterGaussian(Image<Bgr, Byte> image){…}

El resultado lo podemos ver en la siguiente imagen:

Figura 4. Imagen original a la izquierda. Imagen con el filtro de Gauss a la derecha.



* *Filtro de desenfoque blur*

Este filtro se encarga de realizar un desenfoque gaussiano a la imagen, dejando la imagen desenfocada.

Básicamente se realiza como el anterior, simplemente la desviación es mayor y al final queda la imagen desenfocada.

El desarrollo realizado ha sido sencillo ya que está función estaba implementada en EmguCv. La función recibe una imagen y devuelve otra resultante de aplicar el filtro.

/\* FUNCION FILTRO DE DESENFOQUE GAUSSIANO \*/

public static Image<Bgr, Byte> filterBlur(Image<Bgr, Byte> image){…}

El resultado lo podemos ver en la siguiente imagen:

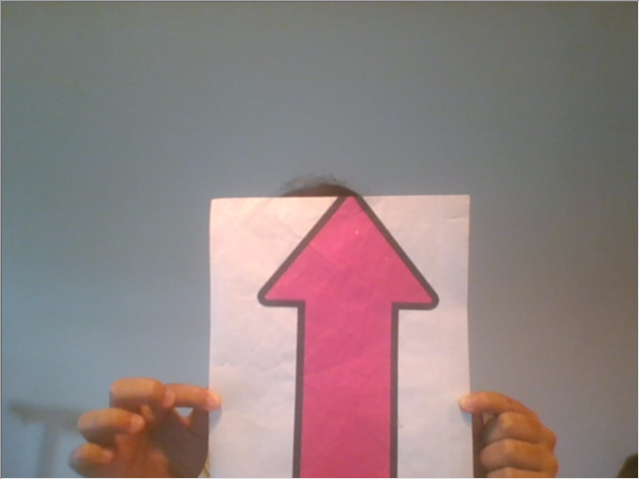


Figura 5. Imagen original a la izquierda. Imagen con el filtro de Blur a la derecha.

* *Filtro Sobel*

El operador Sobel es utilizado principalmente para la detección de bordes. Calcula una aproximación al gradiente de intensidad de la imagen.

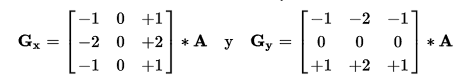
Así, para cada punto, este operador da la magnitud del mayor cambio posible, la dirección de éste y el sentido desde oscuro a claro. El resultado muestra cómo de abruptamente o suavemente cambia una imagen en cada punto analizado y, en consecuencia, cuán probable es que éste represente un borde en la imagen y, también, la orientación a la que tiende ese borde. En la práctica, el cálculo de la magnitud -probabilidad de un borde- es más fiable y sencillo de interpretar que el cálculo de la dirección y sentido.

Matemáticamente, el [gradiente](https://es.wikipedia.org/wiki/Gradiente) de una función de dos variables (en este caso, la función de intensidad de la imagen) para cada punto es un [vector](https://es.wikipedia.org/wiki/Vector) bidimensional cuyos componentes están dados por las primeras derivadas de las direcciones verticales y horizontales. Para cada punto de la imagen, el vector gradiente apunta en dirección del incremento máximo posible de la intensidad, y la magnitud del vector gradiente corresponde a la cantidad de cambio de la intensidad en esa dirección.

Lo dicho en los párrafos anteriores implica que el resultado de aplicar el operador Sobel sobre una región con intensidad de imagen constante es un [vector cero](https://es.wikipedia.org/wiki/Vector_nulo), y el resultado de aplicarlo en un punto sobre un borde es un vector que cruza el borde (perpendicular) cuyo sentido es de los puntos más oscuros a los más claros.

Generalmente se el operador usa dos “*kernels”* o cuadrillas 3x3 para aplicar la convolución a la imagen original, y así aproximar a las derivadas horizontales y verticales. Una convolución es un [operador](https://es.wikipedia.org/wiki/Operador) matemático que transforma dos [funciones](https://es.wikipedia.org/wiki/Funci%C3%B3n_(matem%C3%A1ticas)) ***f***y ***g*** en una tercera función que en cierto sentido representa la magnitud en la que se superponen ***f*** y una versión trasladada e invertida de ***g***.

Si definimos **{\displaystyle \mathbf {A} }A** como la imagen original, el resultado, que son las dos imágenes **Gx** {\displaystyle \mathbf {G\_{x}} } y **Gy** {\displaystyle \mathbf {G\_{y}} } que representan para cada punto las aproximaciones horizontal y vertical de las derivadas de intensidades, es calculado como:



Siendo lo aplicado a cada punto de la imagen:



Esto hará que se noten más las intensidades que sean más oscuras que otras, detectando así los bordes.

El desarrollo realizado ha sido sencillo ya que está función estaba implementada en EmguCv. La función recibe una imagen y devuelve otra resultante de aplicar el filtro.

/\* FUNCION FILTRO SOBEL \*/

public static Image<Bgr, Byte> filterSobel (Image<Bgr, Byte> image){…}

El resultado lo podemos ver en la siguiente imagen:

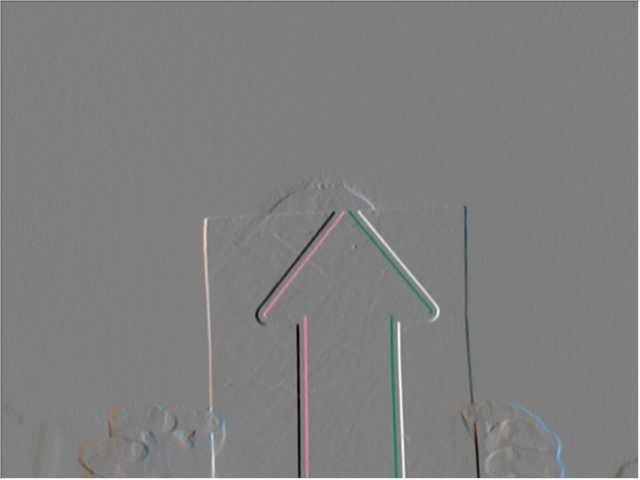
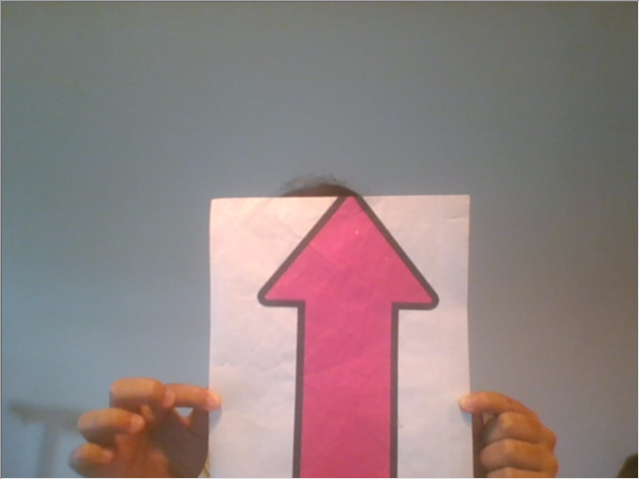


Figura 6. Imagen original a la izquierda. Imagen con el filtro de Sobel a la derecha.

* *Filtro Canny*

El operador de detección de bordes de Canny fue desarrollado por un catedrático de la universidad de Berkeley (EEUU) en 1986 y se basa en un algoritmo de múltiples fases para detectar un amplio rango de bordes. Es sin duda el operador más utilizado en la detección de bordes.

El objetivo de Canny era descubrir el algoritmo óptimo para la detección de bordes. Un detector óptimo significará una buena detección, el algoritmo debe marcar tantos bordes reales como sea posible, una buena localización, los bordes marcados deben estar lo más cerca posible del borde en la imagen real, y por último una mínima respuesta, es decir, un borde dado debe ser marcado sólo una vez y donde sea posible el ruido presente en la imagen no debería crear falsos bordes.

1. **Operador de Canny:**  
   Esta técnica, que se caracteriza por estar optimizada para la detección de bordes diferenciales, consta de 3 fases pero algunos autores agregan una ultima fase, siendo en total 4 fases.
2. **Fases de detección de bordes de Canny:**

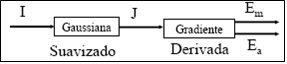
* Obtención del gradiente
* Supresión no máxima al resultado del gradiente
* Histéresis de umbral a la supresión no máxima
* Cierre de contornos abiertos

**Obtención del gradiente:**Para obtener el gradiente se tiene que suavizar la imagen. Se aplica a la imagen **I** un suavizado gaussiano **G** o filtro gaussiano **J** = **I \* G**.

A continuación se calcula el gradiente, mediante la magnitud de los bordes (intensidad):

http://3.bp.blogspot.com/_oLkZ3g-AJQM/SePkA4za2tI/AAAAAAAAAK0/NqzMNVD_M78/s400/Dibujo1.bmp

http://2.bp.blogspot.com/_oLkZ3g-AJQM/SePkqj4SeUI/AAAAAAAAAK8/vp3EhAhV4gA/s400/Dibujo2.bmpPor último se estima la orientación de los bordes:

****

**Supresión no máxima al resultado del gradiente**El objetivo de este paso es obtener bordes de 1 pixel de grosor al considerar únicamente pixels cuya magnitud es máxima en bordes gruesos y descartar aquellos cuyas magnitudes no alcancen ese máximo.  
  
**Histéresis de umbral a la supresión no máxima**Permite eliminar máximos procedentes de ruido, interferencias en el sistema de entrada, etc. Para cada píxel que pase de un umbral definido, se buscarán aquellas cadenas de máximos locales perpendiculares a la normal del borde siempre que sean mayores que otro umbral (el primer umbral será siempre mayor que este último).

La salida será el conjunto de bordes conectados así como la magnitud y la orientación.  
  
**Cierre de contornos abiertos**(Algoritmo de Deriche y Cocquerez)  
También se conoce como el algoritmo de Deriche y Cocquerez. Lo primero es binarizar la imagen en contornos (1 = borde; 0 = no borde). Para cada punto de borde de un extremo abierto se le asigna un código que determina las direcciones de búsqueda para el cierre del contorno.

Para los píxeles marcados con este código se marca como pixel de borde el de máximo gradiente en las tres direcciones posibles. Se repiten los pasos hasta que se cierren todos los contornos.

El desarrollo realizado ha sido sencillo ya que está función estaba implementada en EmguCv. La función recibe una imagen y devuelve otra resultante de aplicar el filtro.

/\* FUNCION FILTRO CANNY \*/

public static Image<Bgr, Byte> filterCanny (Image<Bgr, Byte> image){…}

El resultado lo podemos ver en la siguiente imagen:

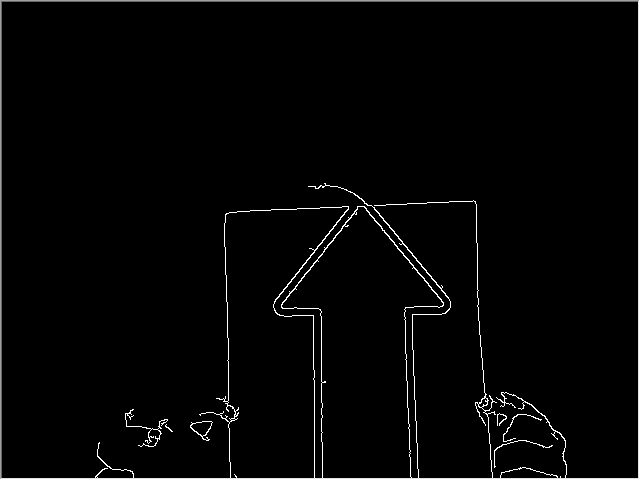
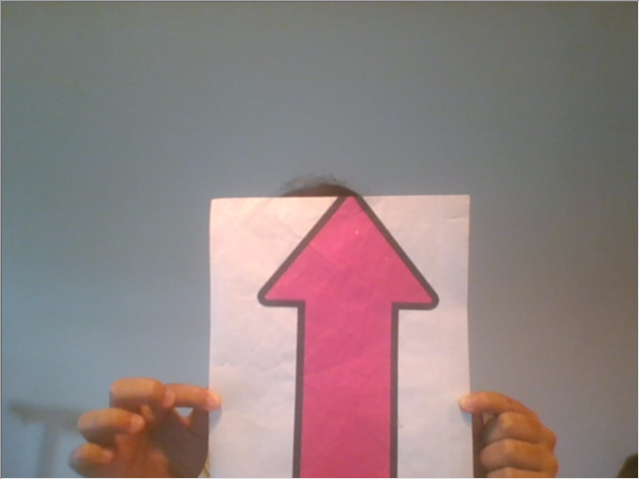


Figura 7. Imagen original a la izquierda. Imagen con el filtro de Canny a la derecha.

* + 1. **Reconocimiento de imágenes: Template Matching**

Las técnicas de template matching se basan en una referencia, conocida como la imagen template o plantilla. Dicha imagen se tratará de identificar dentro de otra imagen, conocida como la imagen source o fuente.

Para ello tratará de encontrar el area de la imagen fuente en donde se encuentre la plantilla. Para exponer mejor esta idea, nos basaremos en la siguiente figura:



Para encontrar la plantilla, el algoritmo, colocará el template dentro de cada posición de la imagen fuente, comprobando la correlación que hay entre ellas.

Figura 8. Template a la izquierda. Source a la derecha.

### Correlación de las imágenes

Uno de los problemas que atañe al template matching es el de calcular la similitud entre las dos imágenes; esto se conoce como correlación entre las imágenes.

Para calcular dicha similitud se usa por tanto el método de *Cross-Correlation*. A continuación podemos ver una tabla de los valores obtenidos con este método.

|  |  |  |
| --- | --- | --- |
| Template | Fuente | Cross-Correlation |
| http://docs.adaptive-vision.com/current/studio/img/machine_vision_guide/template_matching/connector_pattern.png | http://docs.adaptive-vision.com/current/studio/img/machine_vision_guide/template_matching/connector_align1.png | 19404780 |
| http://docs.adaptive-vision.com/current/studio/img/machine_vision_guide/template_matching/connector_pattern.png | http://docs.adaptive-vision.com/current/studio/img/machine_vision_guide/template_matching/connector_align2.png | 23316890 |
| http://docs.adaptive-vision.com/current/studio/img/machine_vision_guide/template_matching/connector_pattern.png | http://docs.adaptive-vision.com/current/studio/img/machine_vision_guide/template_matching/connector_align3.png | 24715810 |

\mbox{Cross-Correlation}(\mbox{Image1}, \mbox{Image2})= \sum_{x,y} \mbox{Image1}(x,y) \times \mbox{Image2}(x,y)La fórmula que sigue la correlación no es más que la suma de la multiplicación de cada píxel de la imagen. Suponiendo que Image1 es la plantilla e Image2 la fuente tendríamos:

Tabla 1. Cross-Correlation.

Estos valores son muy altos por lo tanto se suelen normalizar. A continuación podemos ver la tabla con los valores normalizados.

Tabla 1. Cross-Correlation.

|  |  |  |
| --- | --- | --- |
| Template | Fuente | NCC |
| http://docs.adaptive-vision.com/current/studio/img/machine_vision_guide/template_matching/connector_pattern.png | http://docs.adaptive-vision.com/current/studio/img/machine_vision_guide/template_matching/connector_align1.png | -0.417 |
| http://docs.adaptive-vision.com/current/studio/img/machine_vision_guide/template_matching/connector_pattern.png | http://docs.adaptive-vision.com/current/studio/img/machine_vision_guide/template_matching/connector_align2.png | 0.553 |
| http://docs.adaptive-vision.com/current/studio/img/machine_vision_guide/template_matching/connector_pattern.png | http://docs.adaptive-vision.com/current/studio/img/machine_vision_guide/template_matching/connector_align3.png | 0.844 |

*Normalized Cross-Correlation* (NCC) es la version mejorada del *Cross-Correlation*, que aporta dos mejoras considerables:

Tabla 2. Normalized Cross-Correlation.

* Los resultados son invariantes a los cambios de iluminación.
* La correlación final es un valor entre -1 y 1, lo que permite más fácil su distinción.

La fórmula a seguir será: \mbox{NCC}(\mbox{Image1}, \mbox{Image2})= \frac{1}{N\sigma_1 \sigma_2} \sum_{x,y} (\mbox{Image1}(x,y)-\overline{\mbox{Image1} }) \times (\mbox{Image2}(x,y)-\overline{\mbox{Image2} })}

### Consideración de puntos encontrados

Ahora lo que necesita el algoritmo, es considerar los puntos que son válidos dentro del reconocimiento. Por defecto el umbral será cuando su NCC sea mayor de 0.5, pero esto se puede variar para encontrar un valor mejor.

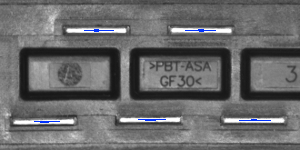
En nuestro proyecto se utiliza el valor por defecto de 0.75.

Figura 8. Template matching con umbral de 0.75.

Este algoritmo es muy lento con imágenes grandes, por lo que generalmente, y en nuestro caso, se usa el procesamiento piramidal que es una técnica de escalado de imágenes para convertirlas en menores.

### Procesamiento piramidal

El procesamiento piramidal redimensiona la imagen a exactamente la mitad, por cada vez que se utiliza. En nuestro caso se ha utilizado tanto para la imagen fuente como para la plantilla dos escalados piramidales, lo que ha permitido que el algoritmo se ejecute de forma veloz.

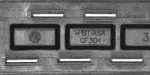


Figura 9. Procesamiento piramidal en dos niveles.

### Problemas de este algoritmo

Como podemos ver, el algoritmo es sensible a cambios de posición de la imagen, es decir a la rotación, también a cambios de tonalidades con los colores y a ruidos externos.

Por ello, muchas veces se suele realizar el algoritmo con un pre-procesamiento de escalado de grises y rotativo. En nuestro caso, la rotación se ha realizado únicamente con el filtro de color aplicado.

Esta rotación se basa en crear una lista de templates que rote a la plantilla original la angulación definida, por lo que al final se realizaría una búsqueda con cada uno de los templates. Esto solo es viable con el procesamiento piramidal, sino el coste en computación sería muy grande.

Como hemos dicho la rotación en nuestra interfaz, solo se admite con el filtro de color.

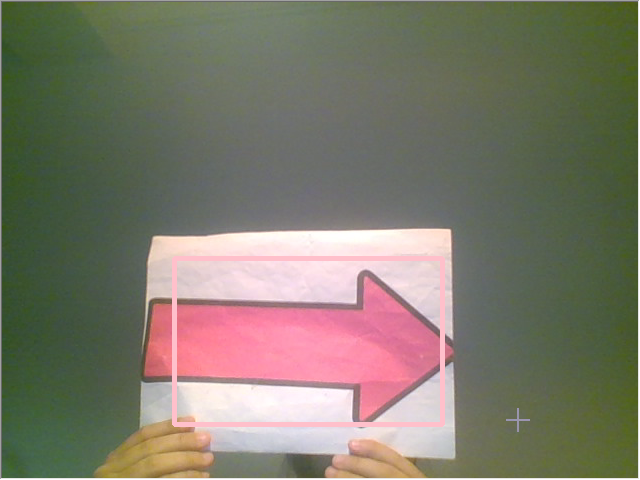
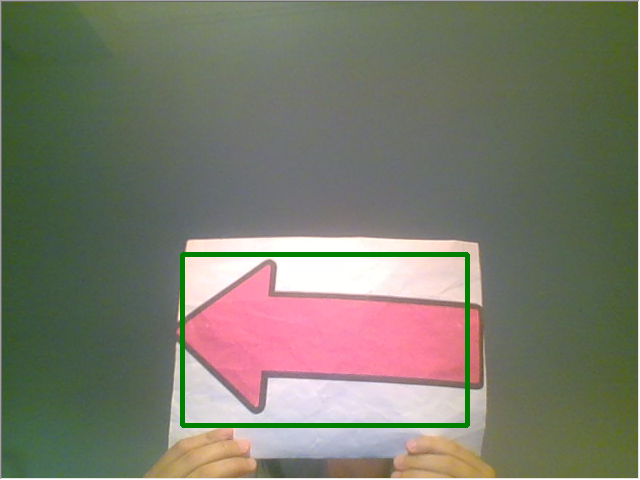
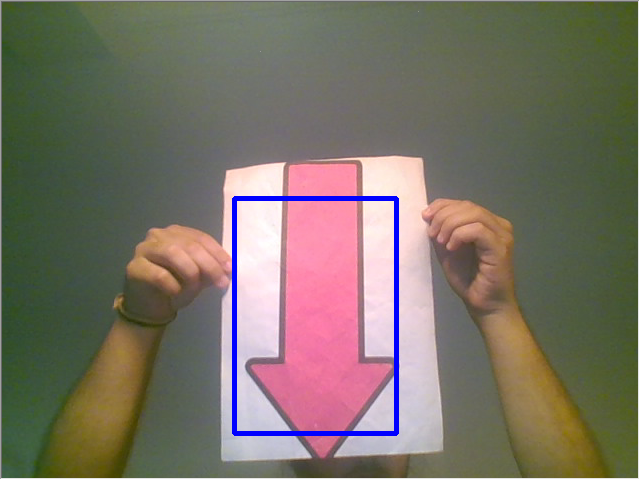
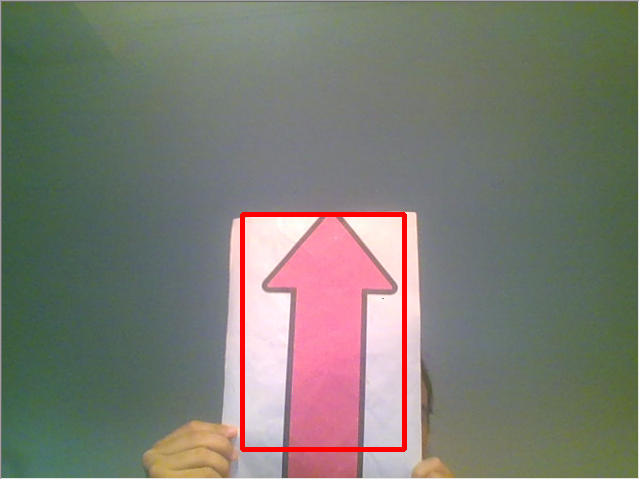


Figura 10. A la izquierda el template. A la derecha reconocimiento rotativo por multiplantilla en template matching.

El desarrollo del reconocimiento, pese a que el template matching básico ya estaba implementado en EmguCv, ha sido costoso pues hemos tenido que añadir el preprocesamiento, la parelización de multiplantillas y algunas funciones auxiliares.

El resultado de nuestro reconocimiento se puede ver en las siguientes imágenes

Figura 11. Reconocimiento en nuestro proyecto según template matching.



/\* VARIABLES \*/

public static Dictionary<string, Image<Bgr, Byte>> templateMap;

public static List<List<Image<Bgr, byte>>> templateMapList;

public static Point maxLocationFinal = new Point();

public static Double maxValueFinal = -1;

public static Image<Bgr, Byte> templateFinal = new Image<Bgr, byte>(1, 1);

public static String templateEncountered = "no encountered";

/\* FUNCION ALGORITMO TEMPLATE MATCHING \*/

public static Boolean templateMatching(Image<Bgr, Byte> source, Image<Bgr, Byte> template, Double threshold){…}

/\* FUNCION INICIO DEL RECONOCIMIENTO \*/

public static void initializeImageRecognition(Dictionary<String, Image<Bgr, Byte>> templates, Boolean filterSaltAndPepperOn, Boolean filterGaussianOn, Boolean filterBlurOn, Boolean filterSobelOn, Boolean filterCannyOn, Boolean filterColorOn, Bgr color){…}

/\* FUNCION PROCESAMIENTO REAL \*/

public static Image<Bgr, Byte> imageProcessing(Image<Bgr, Byte> capture, Double threshold, int typeOfProcessing, Boolean filterSaltAndPepperOn, Boolean filterGaussianOn, Boolean filterBlurOn, Boolean filterSobelOn, Boolean filterCannyOn, Boolean filterColorOn, Boolean showFilters)

{

* + 1. **Otros algoritmos de reconocimiento y detección**

Pese a que el único algoritmo desarrollado ha sido el template matching, hemos estudiado otros dos tipos de reconocimiento.

El primero que intentamos desarrollar fue el de haarscascade, que es un algoritmo que se basa en unos valores de un xml que definen las formas de la imagen. Dicho algoritmo se suele utilizar mucho para reconocer manos, caras, etc. Esto se debe a que no depende de una imagen concreta, sino que usa como plantilla valores de los píxeles y sus píxeles circundantes para encontrar formas.

Era un algoritmo muy interesante a estudiar debido a su potencial, sin embargo, tras varias pruebas coincidimos que tenía demasiados falsos positivos y por otra parte, el coste de generar una plantilla xml con los valores requeridos era bastante alto, por lo que no estudiamos en más profundidad dicho algoritmo.

También intentamos desarrollar un segundo algoritmo de reconocimiento que prometía un potencial importante. Este es el SURF, *Speeded-Up Robust Features*, que se basa en sacar las características de una imagen y compararlas con la plantilla.

Sin embargo, la búsqueda de características tras unas cuantas pruebas, no funciona correctamente con plantillas simples, con lo que decidimos utilizar directamente el template matching.