Trabajo final.

Herramienta para extracción de objetos y pegado de estos en otras imágenes.

Alberto Armijo Ruiz. GII

Introducción.

En esta práctica se desarrollará un herramienta que permita recortar un objeto de una imagen y pegarla en otra imagen, intentando que en la imagen final no se note que el objeto está pegado.

Por lo tanto, se dividirá la explicación del proceso realizado en dos partes, cómo se ha recortado el objeto desde la imagen y cómo se ha pegado el objeto en la imagen de destino y se ha actuado para evitar que se note en la imagen final que está pegado.

También se introducirá un pequeño apartado donde se comentarán posibles mejoras que se podrías hacer a la herramienta que se describe. Por último, se añadirá un apartado con enlaces a la documentación de la funciones usadas.

Recortado de objetos en imágenes.

En esta sección se explicará con detalle el proceso que se ha seguido para obtener un objeto recortado de una imagen, también se mostrarán algunos ejemplos obtenidos en el desarrollo de esta.

Creación de la máscara para recortar.

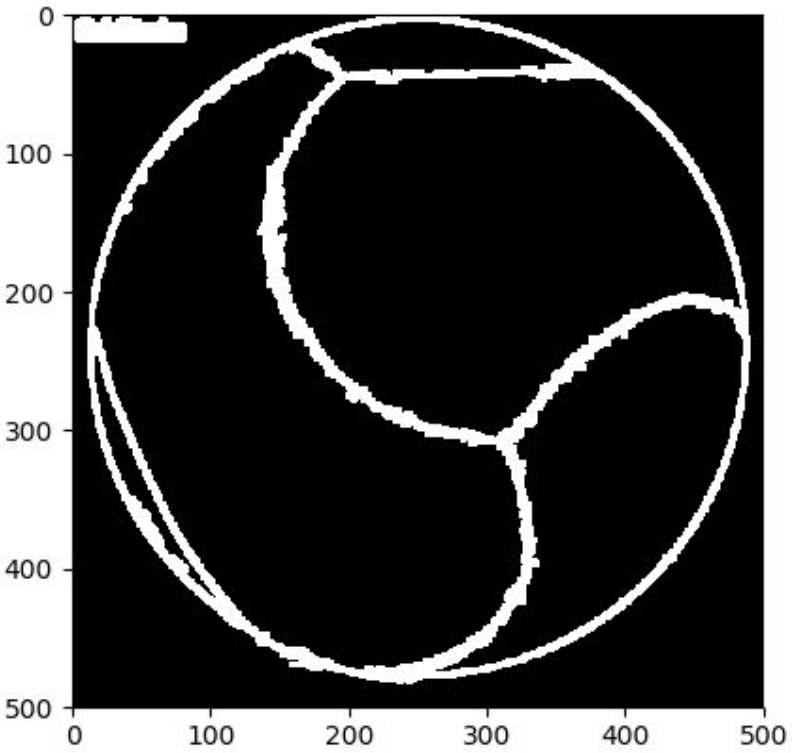
Para crear un máscara para recortar los objetos sobre las imágenes, utilizaremos las funciones Canny, threshold, dilate, erode, findContours y drawContours de OpenCV.

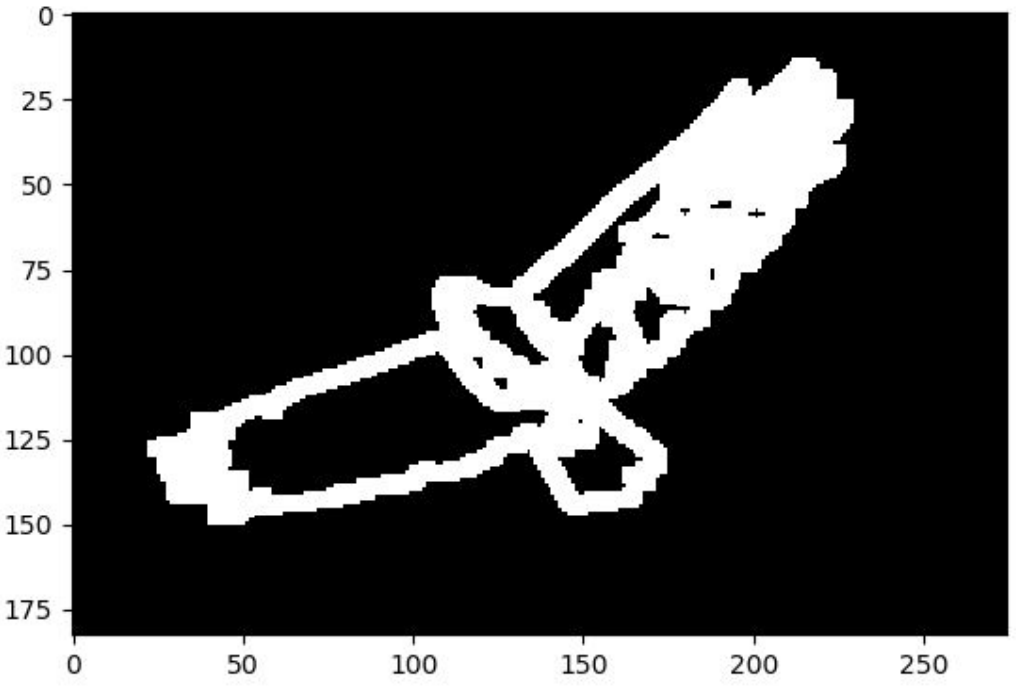
Lo primero que haremos será detectar bordes de nuestro objeto en la imagen, para ello utilizaremos la función cv2.Canny(); esta función implementa el detector de bordes Canny. Lo primero que hace es aplicar un filtro Gaussiano de tamaño 5 a la imagen para reducir el posible ruido que hay en la imagen. Tras esto, se crean los gradientes en x e y de la imagen y se computa la intensidad del gradiente para cada pixel de la imagen, por defecto la función que utiliza OpenCV para calcular esto es:
$EdgeGradiente(grayImg) = |grayImgX| + |grayImgY|$.
Lo siguiente que hace esta función es realizar una supresión de No Máximos, para ello comprueba en cada pixel que se encuentra en un borde, comprueba si es máximo local o no, si no es así, entonces se suprime.Por último, una vez tenemos todos los ejes detectados, nos quedamos con aquellos que se encuentran dentro de un rango que definimos nosotros.

Lo que nos devuelve esta función es un imagen en blanco y negro en la cual los píxeles en blanco son aquellos que forman parte de algún borde. A esta función lo único que es necesario pasarle es la imagen en blanco y negro donde se encuentra el objeto, y un

umbral, dicho umbral será el valor mínimo que se considerará para seleccionar un borde; como valor máximo para considerar un borde se utilizará dos veces el valor del umbral.

Lo siguiente que haremos es utilizar la función cv2.dilate(), esta función aumenta el área que ocupa cada uno de los bordes. Para utilizar esta función, debemos declarar un kernel, para esta práctica, con un kernel de unos de tamaño 5 da buenos resultados ya que solo lo necesitamos para que la función findContours obtenga mejores resultados. Ahora, se les mostrará los resultados obtenidos con una pelota y un aguila.





Lo siguiente que haremos es utilizar la función cv2.threshold(), esta función toma como parámetros la imagen, un umbral, un valor por defecto, y el tipo de threshold. Para esta práctica solo se han utilizado dos tipos de threshold, cv2.THRESH_BINARY y cv2.THRESH_BINARY_INV; uno funciona al revés del otro. Para el primero, se comprueba si el valor de cada pixel es mayor que el umbral especificado, si es así, se establece el valor del umbral al valor especificado como tercer parámetro. Para el segundo, el procedimiento es el inverso, si el valor de un pixel es mayor que el umbral, se establece su valor a 0, si no, se establece al valor especificado. Para la práctica, el umbral siempre será 0 y el valor por defecto será 255, el tipo de threshold utilizado dependerá del objeto que estemos recortando, ya que para algunas imágenes funciona mejor la imagen inversa que la obtenida por Canny.

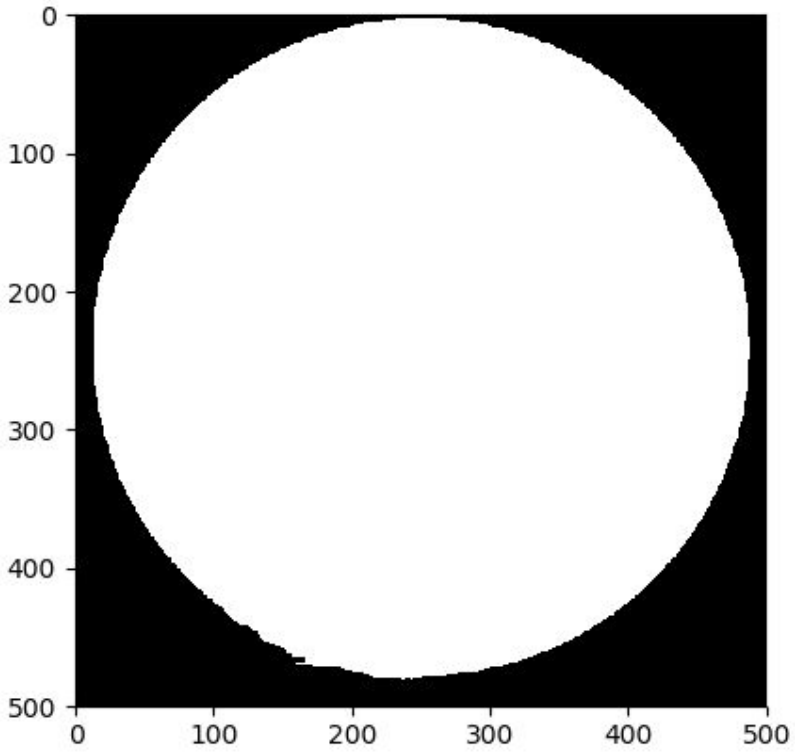
El siguiente paso será detectar los contornos de la imagen, para ello utilizaremos la función cv2.findContours, a esta función siempre le pasaremos la imagen creada por cv2.threshold() y el resto de parámetros utilizados serán cv2.RETR_TREE y cv2.CHAIN_APPROX_SIMPLE. Esta función lo que hace es detectar los contornos dentro de una imagen, es decir, píxeles continuos en el espacio que que tienen un mismo color o una misma intensidad (para el caso de la imágenes en blanco y negro). Lo que nos interesa de esta función es el segundo objeto que devuelve; este es una lista, en la cual cada objeto de la lista es otra lista que contiene los puntos que contiene cada contorno.

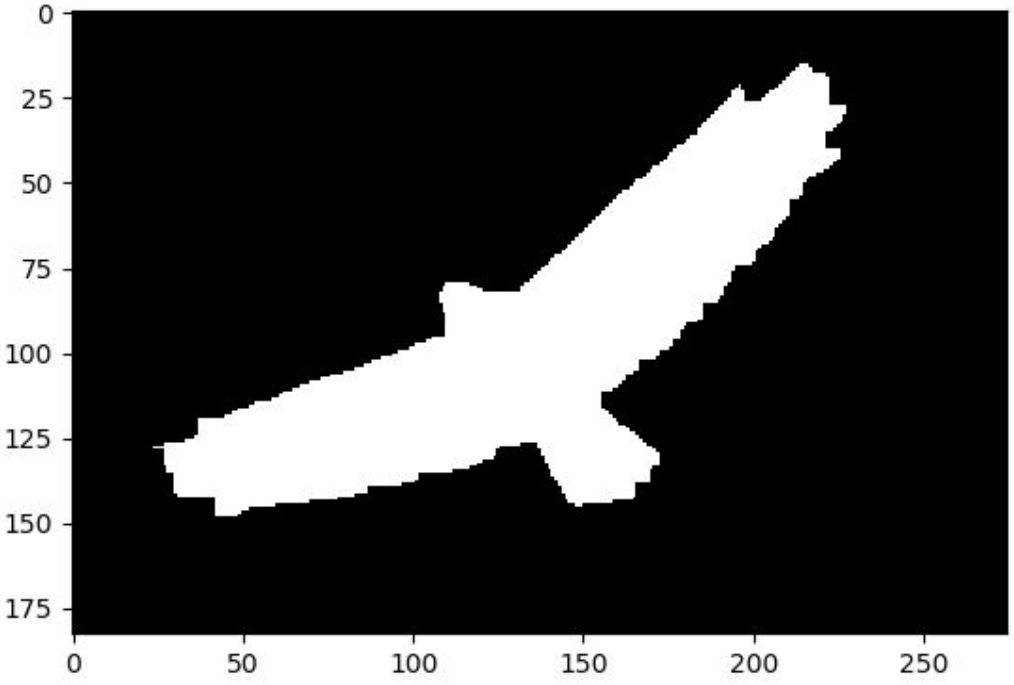
De todos los contornos que hemos encontrado, solo nos interesa quedarnos con el más grande, que es aquel que incluye el objeto entero. Para ello, recorreremos cada uno de los contornos obtenidos, calcularemos su área con la función cv2.contourArea() y nos quedaremos con aquel que tenga mayor área.

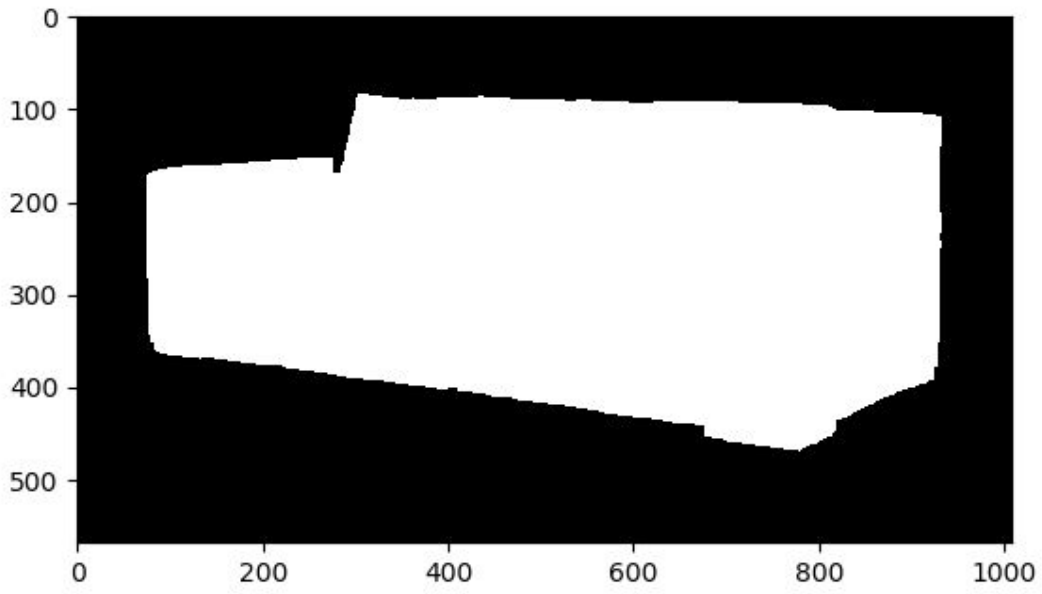
Finalmente, crearemos la máscara con la cual podremos recortar el objeto dentro de la imagen. Para ello, crearemos un imagen entera de ceros con las dimensiones de la imagen. Después, utilizaremos la función cv2.drawContours(), esta función dibuja en una imagen un contorno seleccionado, con un color y un grosor determinados. Nuestros parámetros serán la máscaras inicializada a 0, el contorno que hemos calculado con mayor área anteriormente, el color será blanco (255,255,255) y por último un el grosor en el cual especificaremos cv2.FILLED para que nos dibuje el contorno relleno. Existe un tercer parámetro el cual se utiliza para indicar el índice del contorno utilizado, en el caso de pasarle la lista de contornos entera; si se le pasa un único contorno, este valor debe ser 0.

Tras esto, debemos utilizar la función cv2.erode() sobre la máscara para reducir un poco el contorno que hemos dibujado, ya que al usar la función cv2.dilate(), este es un poco más grande que la forma del objeto que vamos a recortar. La función cv2.erode() hace justamente lo contrario a la función cv2.dilate(). Para esta función utilizaremos el mismo kernel que hemos utilizado en la función cv2.dilate().

Ahora se mostrarán algunos de los resultados obtenidos.


0
100
200
300
400
500
0
100
200
300
400
500


0
25
50
75
100
125
150
175
0
50
100
150
200
250



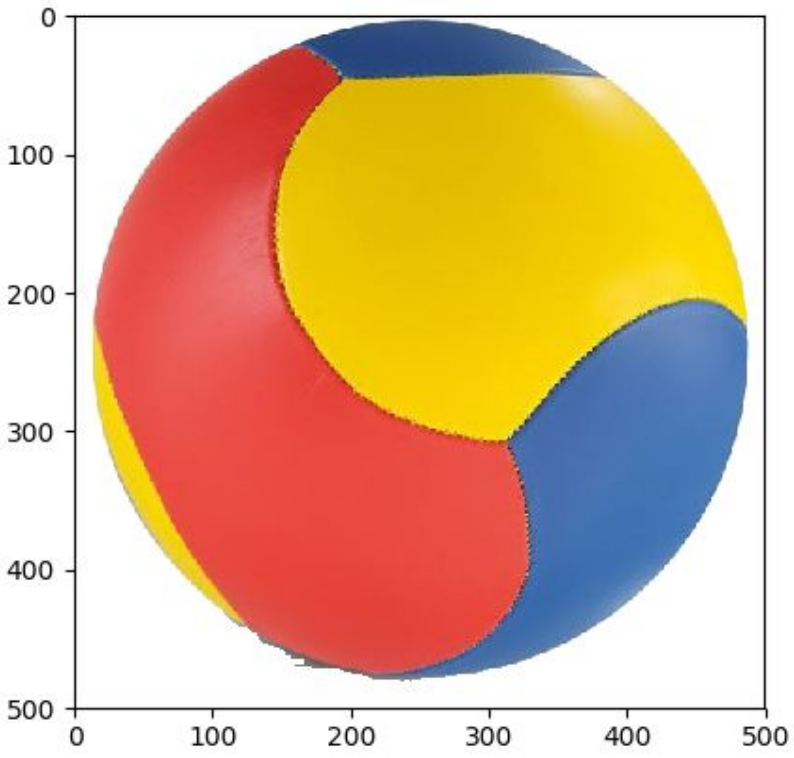
Como se puede ver, las máscaras se adaptan perfectamente a la forma del objeto que queremos recortar. Todo este proceso explicado se realiza en la función extractObjectFromImage(), que devuelve la máscara para recortar el objeto. Debemos de pasarle la imagen, un threshold para la función cv2.Canny(), y el tipo de threshold utilizado para la función cv2.threshold().

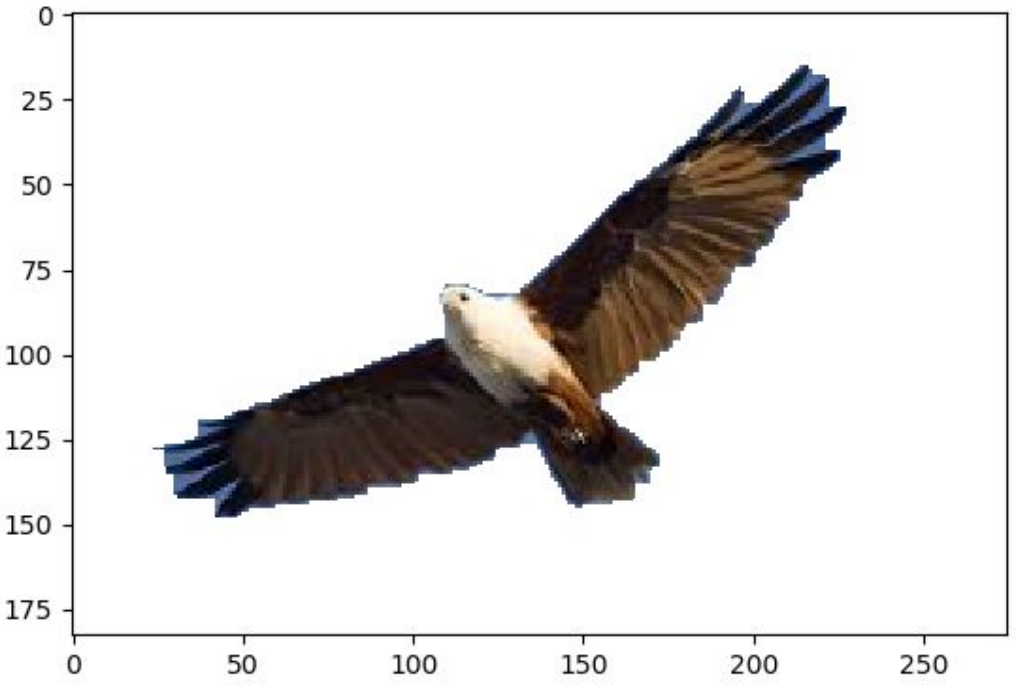
Recortar el objeto de la imágen a partir de la máscara.

Para recortar el objeto de la imagen, se ha creado otra función llamada cropObjectFromImage(), que toma como parámetros la imagen original y la máscara que hemos creado anteriormente.

Esta hace una copia de la imagen original y pone a cero todos aquellos píxeles que sean 0 en la máscara. Después, se crea una capa alpha para que la imagen no se vea con el fondo negro, para ello utilizaremos la máscara y la uniremos a la imagen con la función cv2.merge(). Este último paso no es necesario realmente ya que a la hora de pegar la imagen en otra se utilizará una máscara para seleccionar qué zonas deben utilizarse de la imagen que se va a pegar y cuáles no.

Los resultados obtenidos son los siguientes:





Como se puede ver, el objeto queda bastante bien recortado, sin que se vean ninguno o casi ningún elemento de la imagen donde se ha extraído.

Pegado de imágenes recortadas.

En este apartado, se describirán dos formas que se han implementado para pegar las imágenes y se comentarán los resultados obtenidos. Ambas pegan la imagen de la misma forma, pero la forma de suavizar la imagen pegada para que no se note son diferentes.

Pegado utilizando máscara y alisado con GaussianBlur.

La para este pegado es usar la función cv2.GaussianBlur() sobre una zona de la imagen en la que estamos pegando.

Para esto se han creado dos funciones, una que crea una máscara para aplicar el filtro gaussiano y otra que pega la imagen en la otra y aplica el filtro.

La función que pega la imagen en el otra se llama pasteImage(), esta toma como parámetros la imagen que queremos pegar, la imagen de destino, la posición de inicio donde se pega la imagen y un parámetro llamado “width” que indica el grosor de la máscara que se crea para aplicar el filtro.

Esta máscara utilizará el contorno del objeto que estamos pegando y el parámetro width indica el grosor que se utiliza al dibujar este contorno con la función cv2.drawContours(). La idea es aplicar solamente el filtro a la zona donde se ha pegado la imagen para así disimular que se ha pegado. El parámetro width no debe ser ni muy grande ni muy pequeño, ya que si es muy grande, se notaría en el objeto y alrededor de él que se ha usado el filtro; y si es demasiado pequeño, en la imagen todavía se notaría los bordes de la imagen pegada.

La función pasteImage() comienza calculando una máscara, la cual le indica en qué píxeles debemos pegar la imagen y en qué píxeles no. Después se calcula la zona de la imagen destino donde se va a pegar la imagen recortada, para ello, usamos los valores que hemos pasado como parámetros, que serán nuestro inicio, y le sumamos las columnas y las filas de la imagen recortada. Una vez hecho esto, se pega la imagen recortada en la imagen destino.

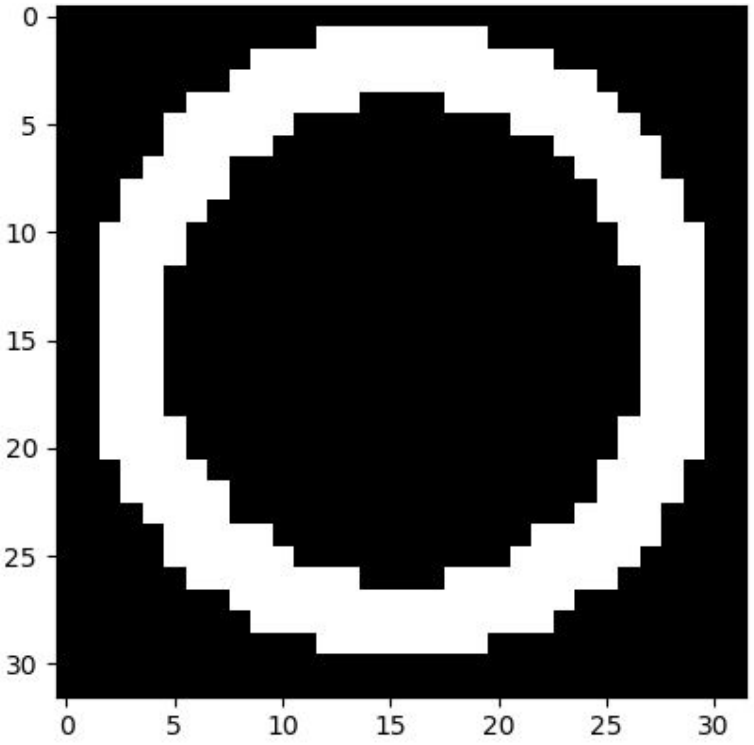
Tras esto, se hace una copia de la imagen combinada, y se le aplica un filtro gaussiano. El tamaño del filtro Gaussiano utilizado es 5 para no provocar demasiada distorsión en la imagen. Tras esto, se llama a la función createMaskForBlurImage(), que se explicará después, pasándole como parámetros el parámetro width y la máscara que hemos creado anteriormente al principio de la función.

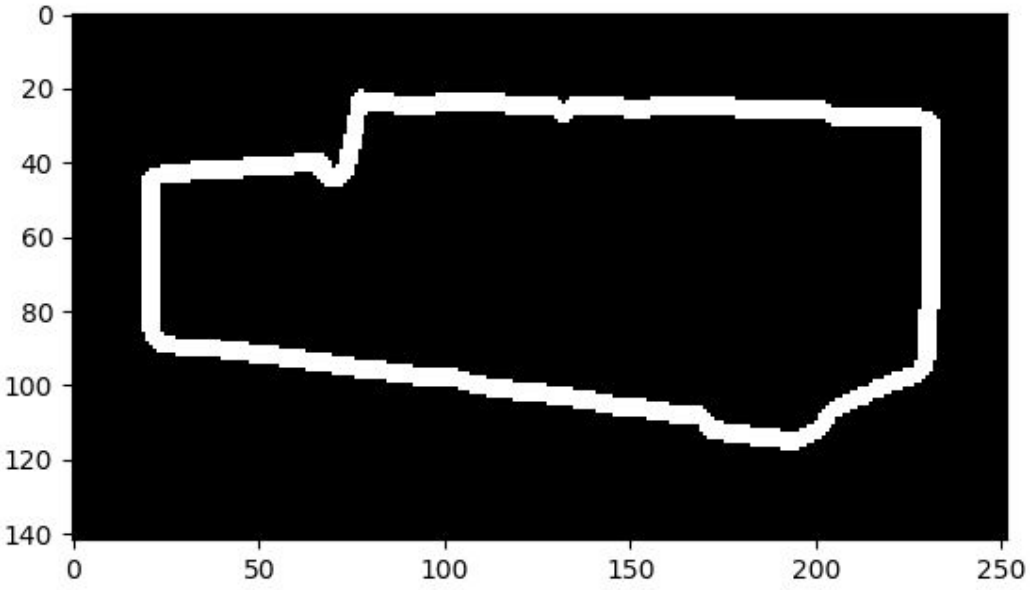
Una vez hemos obtenido esta nueva máscara, debemos actualizar el valor de los píxeles de la imagen combinada por los de la imagen con filtro en aquellos píxeles donde la máscara lo indique. Tras esto devolvemos la imagen final.

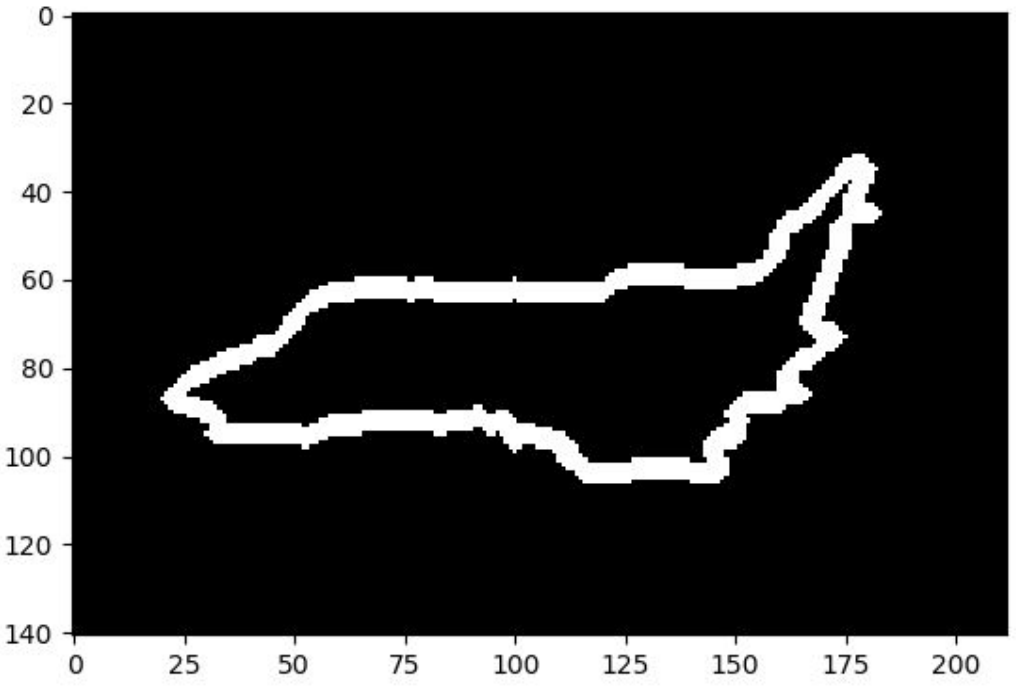
La función createMaskForBlurImage(), simplemente detecta los contornos de la máscara que le hemos pasado, calcula el contorno de mayor área y lo dibuja con el grosor que le especificamos. Tras esto, devolvemos la máscara.

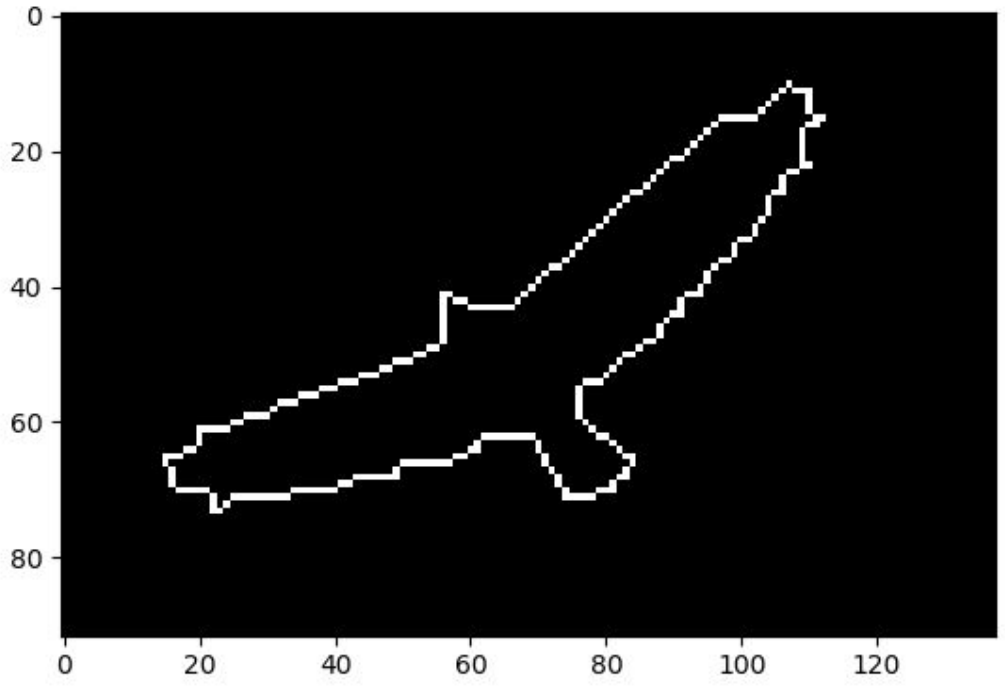
Como algunas imágenes tienen un tamaño muy grande para ser pegadas directamente en las imágenes, se ha reducido el tamaño de estas utilizando la función cv2.pyrDown() hasta obtener un tamaño que sea correcto con respeto a lo que se vé en la imagen.

Los resultados obtenidos son los siguientes:


0
5
10
15
20
25
30
0
5
10
15
20
25
30


0
20
40
60
80
100
120
140
0
50
100
150
200
250


0
20
40
60
80
100
120
140
0
25
50
75
100
125
150
175
200


0
20
40
60
80
0
20
40
60
80
100
120

Como se puede ver, los resultados son en algunos casos bastante convincentes, con esta técnica, se ha conseguido eliminar los bordes de las imágenes pegadas dentro de la imagen destino, que nos haría sospechar rápidamente que el objeto está pegado, sin comprometer demasiado la imagen destino, ya que si los bordes fueran demasiado grandes, se deformaría el fondo de las imágenes destino y también notaríamos que los objetos están pegadas.

El principal problemas que nos encontramos es que si la luz o los colores de los objetos no son muy parecidos a los de las imágenes, también se nota que no pertenecen a las imágenes de destino.

Pegado utilizando pirámides Laplaciana y Gaussiana.

Para este tipo de pegado la idea es muy parecida al proceso que se realiza para suavizar las zonas de intersección entre dos imágenes que se unen en mosaicos.

Nuestra zona de intersección será la zona de la imagen donde vamos a pegar la imagen. Hay una cosa que tenemos que tener en cuenta a la hora de implementarlo ya que este caso no es igual al de los mosaicos, en el cual aparecen los mismo objetos. En nuestro caso, debemos de tener en cuenta que el objeto que vamos a pegar siempre estará encima de la imagen original.

La idea que se quiere utilizar es crear una pirámide gaussiana para la imagen recortada y para la zona de la imagen donde la vamos a crear, el número de niveles dependerá del suavizado que se quiera conseguir, para los ejemplos utilizados siempre es 2. Por cada uno de los niveles de la pirámide laplaciana de ambas imágenes, se creará una última pirámide en la que se pegará la imagen recortada (en ese nivel de la pirámide), en la imagen de destino, esto se hace en la función createPastePyramid(). Una vez hemos obtenido eso, obtendríamos la imagen combinada que vamos a pegar aumentando el tamaño del último nivel de la pirámide que contiene las imágenes combinadas, utilizando la función cv2.pyrUp() especificando el tamaño de la imagen y añadiendo el detalle que tendría en ese nivel, el cual se guarda en los niveles inferiores al último nivel de la pirámide, esto se hace en la función reconstructFromPyramid().

Para combinar la imágenes, se ha creado una función llamada simplePaste() que recibe ambas imágenes y una máscara, que le indica que parte de la imagen recortada tiene que pegar en la imagen destino (en la zona en la que se va a pegar de la imagen, que es con lo que realmente trabajamos en las pirámides). La máscara debe ser siempre del mismo tamaño que la imágen que se va a pegar, por lo tanto debemos tener una pirámide guassiana de dicha máscara.

Para crear pirámides gaussiana y laplaciana se han implementado las funciones createGaussianPyr(), que recibe la imágen y el número de niveles de la pirámide (el primer nivel de la pirámide es siempre la imagen original); y createLaplacianPyr(), que recibe una pirámide gaussiana para crearla.

Para este pegado se ha creado una función llamada pasteImage2(), en esta se crea la máscara que se va a utilizar para pegar el objeto. Tras esto, se crean las pirámides laplacianas de la zona donde se va a pegar de la imagen de destino y de la imagen recortada; también se crea la pirámide de la máscara que hemos creado.

Tras esto se crea la pirámide de imágenes pegadas y se llama a la función reconstructFromPyramid() para obtener la imagen que vamos a pegar. Por último, se sustituye la zona de la imagen donde se va a pegar por la imagen combinada que hemos creado y se devuelve la imagen.

Los resultados obtenidos son los siguientes:

Para este tipo de pegado, los resultados son peores que los obtenidos con el pegado anterior. En algunas se nota perfectamente que está pegada porque se notan los bordes, como es el caso de esta última foto, y en otras se pierde detalle de la imagen pegada por lo cual también se nota que está pegada.

Posibles mejoras.

Existen varias mejoras para esta herramienta. La primera sería intentar automatizar los parámetros que se utilizan para recortar las imágenes, ya que para conseguir un buen resultado hay que ajustar bien dichos parámetros.

La segunda mejora posible sería intentar hacer una mezcla entre los colores que hay en la imagen y los que hay en la imagen que queremos pegar. Es decir, en la primera imagen (la pelota y el niño en la playa) los colores de la pelota no concuerdan con el tono anaranjado que tiene toda la imagen de destino. También intentar cambiar la iluminación que tienen los objetos recortados para que concuerden con la iluminación que hay en las imágenes de destino.

Código de la práctica.

En el código se encuentran todas las funciones, pero solamente se muestran los outputs de las funciones que realizan el pegado de las imágenes.

Enlaces.

https://docs.opencv.org/3.3.1/da/d22/tutorial_py_canny.html
https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_imgproc/py_morphological_ops/py_morphological_ops.html
https://docs.opencv.org/3.3.1/d7/d4d/tutorial_py_thresholding.html
https://docs.opencv.org/3.3.1/d4/d73/tutorial_py_contours_begin.html
https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_imgproc/py_pyramids/py_pyramids.html
http://pages.cs.wisc.edu/~csverma/CS766_09/ImageMosaic/imagemosaic.html