

Laboratorio N° 5- Segmentación y Reconocimiento			
Asignatura	Tópicos Especiales - Visión Artificial	Código	0756
Profesor	José Carlos Rangel Ortiz		

Introducción

En este laboratorio se mostraran lo diferentes algoritmos disponibles en OpenCV para detectar bordes, buscar correspondencias entre imágenes a través de descriptores detectar marcadores, rostros y códigos QR. De igual manera mediante el uso de estos temas se abordará un introducción a los procedimientos de Realidad Aumentada con OpenCV y el uso de Clasificadores en cascada con características de Haar.

Adicional a este documento se facilita una carpeta con los códigos y algunas imágenes y archivos adicionales utilizados en el desarrollo del laboratorio.

La mayor parte de este documento se basa en los ejemplos presentados en el libro : *Mastering OpenCv with Python* de Alberto Fernández Villán [1].

Fundamentos Teóricos

La Visión Artificial busca desde sus inicios imitar el comportamiento de la visión humana. Para ello emplea un conjunto diverso de técnicas que se enfocan en extraer información de imágenes de entrada y así facilitar a las computadoras realizar operaciones como lo hacen los seres humanos.

En Visión Artificial la mayor parte de los algoritmos se basan en investigaciones y descubrimientos anteriores, por lo cual, una gran cantidad de los métodos existentes requieren un pre-procesamiento para poder funcionar. Dentro de estas operaciones podemos mencionar las detecciones tanto de puntos, bordes, líneas, características, rostros o partes del cuerpo. Como podemos ver la detección se puede hacer a diferentes niveles y para cada uno de estos existirán un compendio de técnicas especializadas para lograr su fin.

Adicionalmente, debido a los grandes avances actuales en la Visión Artificial, en los últimos años ha surgido el enfoque de ir más allá de la emulación del humano, sino que se enfoca en enriquecer la percepción visual humana a través de la creación de estructuras virtuales que permitan una interacción del humano con elementos que en realidad no existen en el mundo real.

Dentro de los enfoques mencionados podemos hablar de la **Realidad Virtual** y la **Realidad Aumentada**, la primera enfocada en crear un mundo virtual donde el usuario esta inmerso y en el cual generalmente no se percibe el exterior de manera visual, ni se interactúa con este. En el otro extremo la Realidad Aumenta no busca crear un mundo o espacio aislado de la realidad, por el contrario esta se enfoca en añadir información al mundo real para nutrir las experiencias o interacciones que pueda tener un usuario con el mundo real mediante la incorporación de

animaciones, objetos u otros elementos virtuales que se reaccionan al comportamiento del usuario y al ambiente exterior. Una de las diferencias que se pueden encontrar entre ambas áreas es el tipo de dispositivo que con el que interactúa el usuario (ver figura 1).

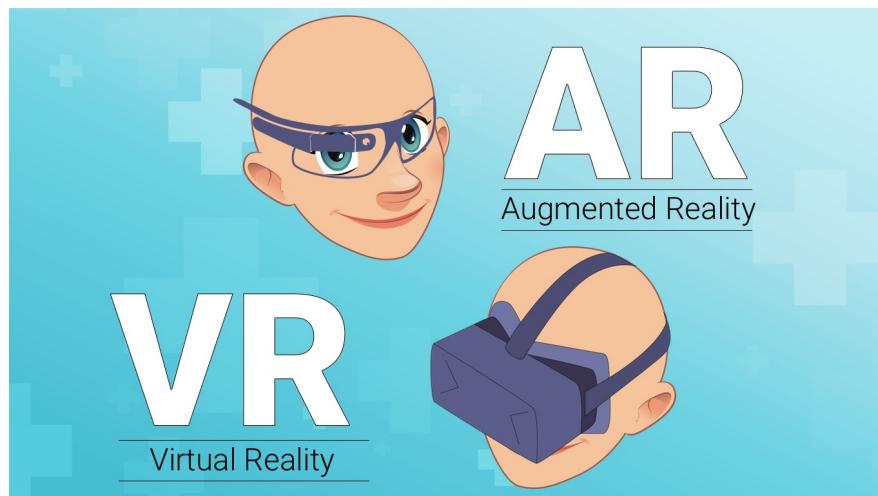


Figura 1: Dispositivos para Realidad Virtual y Realidad Aumentada

La Realidad Aumentada en la actualidad se puede ver aplicada a través de las cámaras de los celulares o *tablets*, en los cuales usando una imagen de entrada de algún lugar permite crear en la aplicación un modelo de un objeto (personas, animales, letras). Puede ser aplicada en comercios, zoológicos, edificios y sus usos en estos lugares son muy diversos, desde permitir visualizar un producto, hasta servir de guía para alcanzar un destino dentro del edificio [2].

En este laboratorio trataremos técnicas utilizadas para aplicar realidad aumentada a la información visual que percibe nuestra cámara. Se enfocará en la utilización de marcadores para añadir información a la realidad y que esta pueda reaccionar a las variaciones que se presentan en el entorno.

Utilizando este enfoque también se presenta una manera de aplicar Realidad Aumentada con sujetos reales mediante la detección de rostros empleando clasificadores en cascada y en base a la detección añadir modificaciones a las caras de estas personas, emulando los tan conocidos filtros disponibles en muchas aplicaciones de redes sociales.

Objetivo

Aplicar los conceptos teóricos de reconocimiento y segmentación a situaciones prácticas que involucren diversos tipos de operaciones utilizando detectores, clasificadores y realidad aumentada.

Entregables

Al final de cada sección del laboratorio se indicará cuales son acciones que deberá cumplir el estudiante y los cuales serán evaluados en la actividad. El informe final debe contener todo lo solicitado y se debe redactar utilizando la Guía de Estilos para Memorias y Laboratorios Disponible en Moodle. El informe debe ser entregado en formato PDF y siguiendo las indicaciones para nombrar el archivo.

1. Emparejamiento de Características

En esta sección nos enfocaremos en la utilización de las características detectadas y descritas utilizando algún descriptor de los disponibles en OpenCV. El emparejamiento consiste en aplicar a un objeto un algoritmo descriptor y luego pasar a una escena el mismo algoritmo con la intención de describir los puntos claves detectados utilizando el mismo proceso.

En este caso utilizaremos el código en el archivo `01_feature_matching.py`, a continuación procedemos a explicar los fragmentos de código relevantes para la ejecución del programa. Este código utiliza como descriptor ORB y como algoritmo de *matching* *Brute Force Matcher*, ambos disponibles en la versión básica de OpenCV.

1. En el siguiente fragmento se configuran ciertos parámetros para la creación de un lienzo donde se colocarán los resultados obtenidos por el método, consiste en una ventana con un color de fondo y una dimensión fijada.

```
# Crear un lienzo con plt para la visualización de los resultados
fig = plt.figure(figsize=(8, 6))
plt.suptitle( "ORB descriptors and Brute-Force (BF) matcher",
              fontsize=14,
              fontweight='bold')
fig.patch.set_facecolor('silver')
```

2. Como siguiente punto se procede a instanciar el *matcher* que utilizaremos en nuestro programa. En esta se debe indicar la formula que se utilizará para calcular la distancia entre los *keypoints* de las 2 imágenes. En el caso de ORB al ser un descriptor binario se utiliza la distancia de Hamming, la cual cuenta la cantidad de bits diferentes entre los descriptores.
3. El segundo parámetro `crossCheck=True`, se activa cuando deseamos que, para un par de características se emparejen en ambos sentidos. Esta opción está predefinida como `False`

```
# Creación de la instancia del matcher
bf_matcher = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
```

4. Como siguiente paso se deben enviar al *matcher* el conjunto de descriptores de ambas imágenes, estas deben estar descritos usando el mismo algoritmo descriptor. El cual generará una lista de características emparejadas.

```
# Iniciar el emparejamiento
bf_matches = bf_matcher.match(descriptors_1, descriptors_2)
```

5. El siguiente punto consiste en ordenar estos emparejamientos de acuerdo a la distancia entre cada uno de los elementos del par. En este caso debemos tener en cuenta que una distancia menor significa mayor similitud entre los pares de características.

```
# Ordenar los pares de acuerdo a la distancia/similitud
bf_matches = sorted(bf_matches, key=lambda x: x.distance)
```

6. Como ultimo paso se procede a identificar en las imágenes originales los pares de características, señaladas con una línea entre estos.

```
# Dibujar los pares de características en las imágenes originales
result = cv2.drawMatches( image_query,
                        keypoints_1,
                        image_scene,
                        keypoints_2,
                        bf_matches[:20],
                        None,
                        matchColor=(255, 255, 0),
                        singlePointColor=(255, 0, 255),
                        flags=0)
```

7. Luego de la ejecución se obtendrá el siguiente resultado (figura 2).

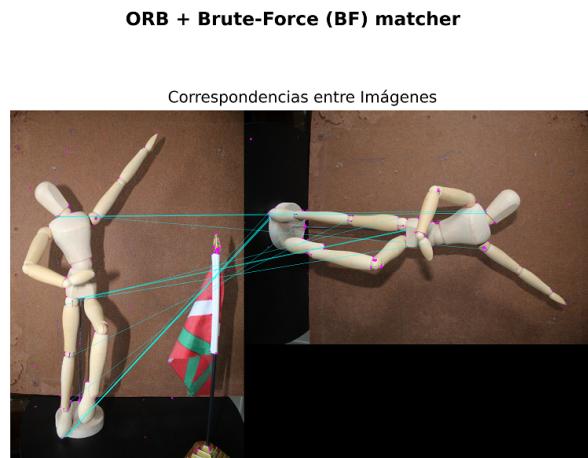


Figura 2: Salida producida por el algoritmo de *matching*.

1.1. Homografía

Una de las utilidades de los algoritmos de correspondencias entre imágenes es la detección de objetos. Por lo cual, una vez encontradas las correspondencias entre un grupo de puntos,

el siguiente paso es encontrar la transformación de perspectiva entre la ubicación de los *key-points* emparejados en las 2 imágenes, lo cual es posible en OpenCV con el uso de la función `cv2.findHomography()`. Una homografía se puede definir como la transformación que determina una correspondencia entre 2 planos o imágenes. Es decir la modificación que se requiere aplicar a un elemento de una imagen para que corresponda con lo observado en otra imagen.

En este caso utilizaremos el código `02_feature_matching_object_recognition.py` dentro de los archivos del laboratorio.

1. En este caso se le envía a la función `findHomography()`, los puntos de la imagen del modelo y de la escena en donde se ubican las características emparejadas. Se indica de igual manera el método que se utilizará para calcular la matriz en este caso se usará el algoritmo de RANSAC¹. El ultimo elemento es un valor del error de proyección para RANSAC.

```
# Encontrar matriz de homografía
M, mask = cv2.findHomography(pts_src, pts_dst, cv2.RANSAC, 5.0)
```

2. Una vez calculada la matriz de homografía, calcularemos las 4 esquinas del objeto, para obtener su representación en la escena.

Esto se realiza con la función `cv2.perspectiveTransform()`, la cual aplica la matriz de homografía al objeto y así se determina su posición en la escena.

```
# Usar M para obtener las esquinas del objeto detectado, en la escena.
pts_corners_dst = cv2.perspectiveTransform(pts_corners_src, M)
```

3. Como ultima acción procedemos a marcar el objeto identificado en la escena.

```
# Dibujar las esquinas del objeto detectado
img_obj = cv2.polyline(image_scene,
                       [np.int32(pts_corners_dst)],
                       True,
                       (0, 255, 255),
                       10)
```

4. Luego de la ejecución se obtendrá el siguiente resultado (figura 3).

¹RANSAC: Random Sample Consensus



Figura 3: Salida producida por el algoritmo de *matching* para reconocimiento de objetos.

Entregables para la Sección

Para esta sección en su informe debe utilizar imágenes de su propiedad. En estas se deben realizar las siguientes operaciones:

1. Pruebe el algoritmo de emparejamiento de características utilizando un descriptor diferente (Brisk o Kaze). Utilice 2 imágenes diferentes en las cuales este presente una instancia del mismo objeto y muestre las correspondencias encontradas. Guarde la imagen resultante de la ejecución de este código y agréguela a su informe.

2. Realidad Aumentada

En esta sección aplicaremos métodos que empleando conceptos de reconocimiento de patrones hacen posible realizar operaciones básicas de realidad aumentada en nuestras imágenes. Los códigos y guías para esta sección se han basado en los presentados en [1].

En esta sección se indicaran los *scripts* que debe ejecutar, estos se encuentran dentro de los archivos suministrados para el laboratorio. En cada sub-sección se explicará la porción de código de mayor relevancia para el cumplimiento de la función deseada. En la mayoría de estos ejemplos se hace necesario la utilización de una cámara web y la impresión de algunos marcadores para demostrar el funcionamiento del código.

2.1. Creación de Marcadores (Markers)

Un elemento importante que nos permitirá aplicar conceptos de realidad aumenta serán los marcadores o *markers* y los diccionarios de estos. Estos están compuestos de un conjunto de celdas internas y externas. Las externas se pintan de negro y forman un borde que puede ser detectado rápida y robustamente. Las celdas internas se utilizan para codificar el marcador. Los marcadores pueden ser de diversos tamaños y cuando se habla de su dimensión hace referencia al tamaño de las celdas internas de la matriz.

En ArUco, se maneja el concepto de diccionarios de marcadores, estos son el conjunto de marcadores que serán utilizados en una aplicación específica. Este módulo permite una manera automática de generación de marcadores con la cantidad de marcadores y la cantidad de celdas que deben tener. De igual manera incluye un conjunto de diccionarios predefinidos para utilizar en nuestras aplicaciones.

Para utilizar los marcadores en nuestro código, debemos, como primera instancia definir el diccionario que utilizaremos y en este ejemplo seleccionar un primer marcador para que se muestre en pantalla, se guarde en disco, se pueda imprimir y así usarse en otras secciones. De igual manera estos marcadores pueden ser reconocidos en una pantalla de tablet, TV o celular.

En esta sección utilizaremos el código `01_aruco_create_markers.py`. En este nos encontramos con el código que permite usar un directorio y guardarlo en disco. Dentro de los posibles diccionarios se pueden mencionar:

- `DICT_4X4_50 = 0`
- `DICT_4X4_100 = 1`
- `DICT_4X4_250 = 2`
- `DICT_4X4_1000 = 3`
- `DICT_5X5_50 = 4`

- DICT_5X5_100 = 5
- DICT_5X5_250 = 6
- DICT_5X5_1000 = 7
- DICT_6X6_50 = 8
- DICT_6X6_100 = 9
- DICT_6X6_250 = 10
- DICT_6X6_1000 = 11
- DICT_7X7_50 = 12
- DICT_7X7_100 = 13
- DICT_7X7_250 = 14
- DICT_7X7_1000 = 15

1. Como primer paso seleccionamos el diccionario, en nuestro caso utilizaremos el diccionario **DICT_7X7_250**, este tiene una dimensión de 7×7 ($n = 7$) celdas y cuenta con un total de 250 marcadores diferentes. El número después del signo =, indica el índice del diccionario en la librería. Para más información de los diccionarios puede consultar la documentación de este módulo en la web².
2. Esta selección de diccionario se realiza en el siguiente fragmento de código:

```
aruco_dictionary = cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT_7X7_250)
```

3. Posterior a esto se puede utilizar la función `cv2.aruco.generateImageMarker()` para obtener un marcador listo para impresión. Utilizando el código a continuación, este recibe como parámetros el diccionario a utilizar, el *id* del marcador deseado, el tamaño en píxeles que se desea dar al elemento (*sidePixels*) y como último elemento la cantidad de celdas que se utilizarán en el borde del marcador.

```
aruco_marker_1 = cv2.aruco.generateImageMarker(  
    dictionary=aruco_dictionary,  
    id=2,  
    sidePixels=600,  
    borderBits=1)
```

4. Ejecute este script desde la consola, utilizando el siguiente código, o mediante su IDE elegido.

²Diccionarios en ArUco

```
python 01_aruco_create_markers.py
```

5. Luego de su ejecución tendrá en su carpeta (donde haya guardado este script) un conjunto de 3 imágenes que contienen un marcador. Para las siguientes secciones debe imprimir el marcador con el nombre `marker_DICT_7X7_250_600_1.png`, puede imprimirla en una hoja tamaño carta o utilizando solo la mitad de esta. El marcador debe verse como en la figura 4 y este caso corresponde con el marcador con $id = 2$ en el diccionario seleccionado. Puede cambiar los $id's$ e imprimir diversos marcadores y así probar los algoritmos con diferentes marcadores. También puede utilizar el siguiente enlace para obtener marcadores Aruco específicos <https://chev.me/arucogen/>

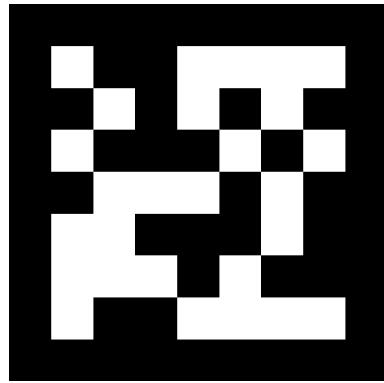


Figura 4: Marcador generado.

2.2. Detección de Marcadores

En esta sección utilizaremos el código `02_aruco_detect_markers.py`, para detectar marcadores en una imagen que se está obteniendo desde una cámara conectada al computador.

1. Como primer paso debemos indicar cual es el diccionario que utilizaremos en la aplicación, por lo cual, serán los marcadores de este diccionario los que podrá identificar nuestro código. Lo cual se hace mediante el siguiente código, se utiliza en este caso un diccionario de los pre-definidos en ArUco, debe coincidir con el diccionario utilizado para crear el marcador impreso en la sección anterior. Se debe también definir el detector el cual utilizará el diccionario seleccionado e indicar los parámetros para el mismo, estos según la necesidad que se tenga de realizar procesos adicionales al procesar la imagen de entrada. [Más información de los Parámetros](#).

El siguiente código se encarga de la selección del diccionario y la creación de la instancia del detector que será utilizado posteriormente.

```
# Definimos nuestro diccionario
dictionary = cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT_7X7_250)
parameters = cv2.aruco.DetectorParameters()
detector = cv2.aruco.ArucoDetector(dictionary, parameters)
```

2. En este caso se utiliza el método `detectMarkers()` del detector creado para procesar una imagen de entrada y buscar marcadores en la misma, esta función recibe como parámetros, una imagen en escala de grises para ser procesada.

```
# Detectamos los marcadores en una imagen de entrada
corners, ids, rejected_corners =detector.detectMarkers(gray_frame)
```

3. Este método devuelve:

- Una lista de las esquinas de los marcadores detectados
- Una lista de los ID's de los marcadores detectados
- Una lista de las esquinas que fueron detectadas, pero que no corresponden con ningún marcadores (lista de rechazados).

4. Luego de la ejecución del script, se identificaran los marcadores y sus ID's en la pantalla y se marcarán los elementos rechazados en la imagen. Para la ejecución escriba lo siguiente en una consola:

```
python 02_aruco_detect_markers.py
```

5. Para cerrar la ventana de visualización puede presionar la letra 'q' en su teclado.

2.3. Calibración de la Cámara

Un proceso de calibración de una cámara consiste en calcular los parámetros internos y externos de la misma. Estos definen el modelo de la misma tanto lineal como no lineal. En la actualidad existen numerosos métodos de calibración que utilizan diferentes tipos de patrones o plantillas, así como también existen los métodos que no hacen uso de las mismas [3]. Estos métodos se enfocan en utilizar patrones con dimensiones conocidas y tomando en cuenta dichas dimensiones calcular los parámetros del dispositivo. La calibración permite, entre otras cosas, obtener resultados más precisos cuando se realicen cálculos de distancias en las imágenes obtenidas por la cámara.

En nuestro caso ArUco cuenta con una función que permite calibrar las cámaras mediante el uso de un conjunto de marcadores de un diccionario seleccionado. Este es un procedimiento que se realiza solo una vez debido a que la óptica de la cámara no será modificada en ningún momento, por lo cual, el resultado obtenido se puede seguir utilizando en las demás partes del laboratorio donde se necesiten. El resultado de la calibración será un archivo serializado con la información obtenida por la función `cv2.aruco.calibrateCameraCharuco()`.

En ArUco para realizar la calibración se utilizan un conjunto de esquinas de varios puntos de vista, extraídos de un tablero con marcadores.

La función de calibración devuelve la matriz de la cámara (3×3 de punto flotante), esta matriz contiene la distancia focal y el centro de coordenadas de la cámara llamados también parámetros intrínsecos. Como segundo elemento la función devuelve un vector que contiene los coeficientes de distorsión, estos modelan la distorsión que es producida por la cámara.

En este caso utilizaremos el código `03_aruco_camera_calibration.py` disponible en los archivos del laboratorio.

1. Como primer paso para el proceso de calibración debemos construir el patrón de calibración que utilizaremos para nuestra cámara. Se define un tablero (similar al ajedrez) con unas dimensiones pre-definidas.
2. Esto se realiza mediante la función `cv2.aruco.CharucoBoard`. En esta función definimos cuantos espacios/bloques/casillas tendrá nuestro patrón de calibración (3 filas y 3 columnas en nuestro caso), las longitud de cada celda de esta matriz (ancho en metros), la longitud de cada marcador en cada celda (ancho en metros) y también el diccionario de donde se obtendrán los marcadores para construir el patrón. Esta función toma los 4 primeros elementos del diccionario para crear la plantilla.
3. Para definir el diccionario, patrón y guardarlos en disco, se emplea el siguiente código:

```
# Definir el diccionario y crear el tablero para calibración
dictionary = cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT_7X7_250)
board = cv2.aruco.CharucoBoard((3,3), .053, .026, dictionary)

# Crear la imagen que contendrá el patrón
image_board = board.draw((200 * 3, 200 * 3))

# Guardar la imagen
cv2.imwrite('charuco.png', image_board)
```

4. La función para crear el tablero, recibe las dimensiones (cantidad de filas y columnas) que debe tener el tablero que crearemos, las dimensiones de un cuadrado del tablero (en metros), dimensión del marcador (en metros) y el diccionario que se utilizará.

5. Para guardar nuestro patrón debemos ejecutar este código de la siguiente manera:

```
python 03_aruco_camera_calibration.py
```

6. La primera ejecución guardará la imagen `charuco.png` en nuestra carpeta y a continuación luego de activar la cámara es probable que se produzca un error ya que no se está detectando ningún patrón.

7. La imagen en disco debe tener la apariencia similar a la mostrada en la figura 5.

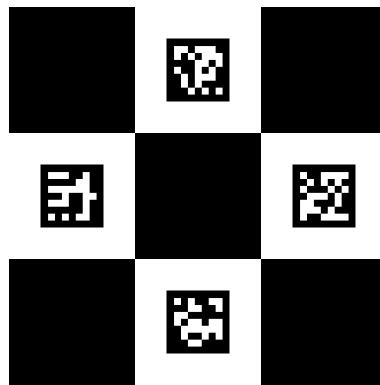


Figura 5: Patrón con marcadores generados para calibración.

8. Como siguiente punto debemos entonces imprimir este patrón cuidando que se mantengan las dimensiones definidas. Si al imprimir y medir las dimensiones reales del panel de calibración no concuerdan con las expresadas en el código, se recomienda modificar el código agregando los valores reales medidos en el patrón impreso.
9. A continuación con el patrón debemos ejecutar nuevamente el código de calibración y enfocarlo hacia nuestra cámara.
10. Durante esta ejecución se pueden presentar errores ya que pueden ocurrir muchas situaciones cuando se está captando el patrón y calculando los parámetros.
11. El código que almacena los parámetros es el siguiente, se emplea la librería de serialización `pickle`

```
# Obtener resultados de calibración:
retval, cameraMatrix, distCoeffs, rvecs, tvecs = cal

# Almacenar parámetros de la cámara:
f = open('calibration.pckl', 'wb')
pickle.dump((cameraMatrix, distCoeffs), f)
f.close()
```

12. Cuando el código se ha ejecutado sin problemas en su carpeta debe estar el archivo `calibration.pckl` el cual contendrá los parámetros calculados para la cámara.
13. Los parámetros de calibración son únicos para cada cámara, por lo cual, si usamos un archivo de calibración generado por otro equipo, es muy probable que se presenten errores al momento de hacer los cálculos.

14. Debe tomar en cuenta que cámara utilizará, en el caso de que tenga varias conectadas al computador. Nuestro código captura *frames* en tiempo real desde la cámara seleccionada y los proyecta en pantalla. Para seleccionar que cámara utilizar se puede modificar el argumento en la siguiente función, el 0 indica la cámara empotrada para laptops, en este caso si usamos una segunda *webcam* debemos colocar el número 1 como argumento.

```
# Definir cámara
cap = cv2.VideoCapture(0)
```

15. Una vez ejecutado el programa 2 veces y obtenidos los parámetros, se puede proseguir con los siguientes ejemplos.

2.4. Detección de Pose de la Cámara

En la realidad aumentada (AR, por sus siglas en inglés) el procedimiento común consiste en superponer un objeto sobre un marcador que ha sido detectado en la imagen. Para ello OpenCV necesita conocer cual es la pose del marcador detectado respecto a la cámara que se está utilizando. Se utilizará el archivo *04_aruco_detect_markers_pose.py*.

En este caso se utilizan los parámetros internos de la cámara calculados en el paso anterior y almacenados en el archivo *calibration.pckl*, para determinar la pose de un marcador que está presente en una imagen captada desde una *webcam*.

Recuerde que si usa más de una cámara debe definir el índice de la cámara que utilizará y la cual fue calibrada anteriormente.

1. En este caso nuestra primera acción será indicarle a OpenCV los parámetros de calibración de nuestra cámara por lo cual se carga y des-serializa el archivo *calibration.pckl*.

```
# abrir archivo con la información de la cámara
f = open('calibration.pckl', 'rb')
cameraMatrix, distCoeffs = pickle.load(f)
f.close()
```

2. De igual manera al utilizar marcadores debemos definir el diccionario de marcadores que utilizaremos para que el programa sepa que está buscando y se crea el detector que hace uso del diccionario.

```
dictionary = cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT_7X7_250)
parameters = cv2.aruco.DetectorParameters()
detector = cv2.aruco.ArucoDetector(dictionary, parameters)
```

3. Luego el procesamiento de las imágenes de entrada incluye el código para detección de marcadores

```
corners, ids, rejectedImgPoints =detector.detectMarkers(gray_frame)
```

4. Finalmente el siguiente código analiza cada uno de los marcadores identificados y devuelve el vector de rotación(rvecs) y de traslación(tvecs) para cada marcador. Esta función recibe:

- a) Esquinas detectadas de los marcadores
- b) Tamaño de un lado de cada marcador (en metros), esto corresponde con el tamaño real de nuestro marcador, se puede dejar en 1m, ya que se utiliza para ayudar a establecer el tamaño de los ejes.
- c) Parámetros de la cámara
- d) Coeficiente de distorsión de la cámara

```
# Obtención de Vectores de rotación y traslación
# de los marcadores detectados
rvecs, tvecs, _ = cv2.aruco.estimatePoseSingleMarkers( corners,
                                                       1,
                                                       cameraMatrix,
                                                       distCoeffs)
```

5. De igual manera se procede a dibujar los ejes sobre cada marcador encontrado utilizando la siguiente instrucción, el ultimo parámetro es el tamaño que deseamos definir para los ejes, esta relacionado con el tamaño real de los marcadores y se define en metros:

```
# Dibujar los ejes
cv2.aruco.drawAxis(frame, cameraMatrix, distCoeffs, rvec, tvec, 1)
```

6. Para ejecutar este programa escribimos la siguiente orden en la consola

```
python 04_aruco_detect_markers_pose.py
```

7. Luego de su ejecución los ejes dibujados se verán de la siguiente manera en nuestra imagen (figura 6):

2.5. Realidad Aumentada con Marcadores

En esta parte seguiremos utilizando los parámetros de calibración de la cámara para superponer un cuadrado a un marcador identificado en la imagen de entrada. Se utilizará el archivo 05_aruco_detect_markers_square.py.

En este caso el código se basa en el anterior, pero con la diferencia que no dibujaremos un conjunto de ejes sobre el marcador, sino un cuadrado en el marcador. Este cuadrado deberá coincidir con las dimensiones del marcador detectado.

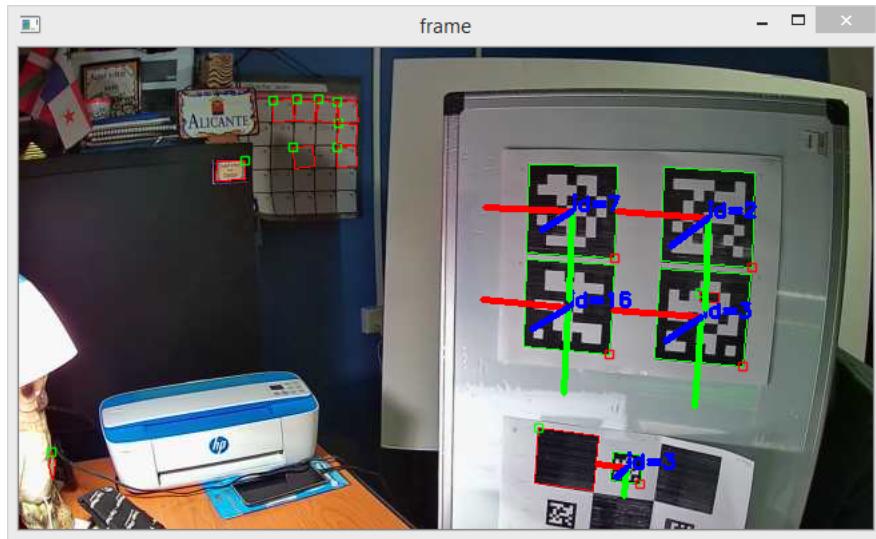


Figura 6: Ejes dibujados en los marcadores detectados.

1. Luego de detectar los marcadores y sus esquinas, se procede a definir los puntos para los cuales deseó obtener las coordenadas para dibujarlos sobre el marcador, lo cual se hace con el siguiente código:

```
# Puntos deseados para superponer en el marcador
desired_points = np.float32( [ [-1 / 2, 1 / 2, 0],
                               [1 / 2, 1 / 2, 0],
                               [1 / 2, -1 / 2, 0],
                               [-1 / 2, -1 / 2, 0]]
                               ) * OVERLAY_SIZE_PER
```

2. El código anterior define las 4 esquinas del cuadrado que queremos dibujar, debemos tener en cuenta que estas no son las coordenadas, sino que representan la ubicación de los puntos con respecto al centro del marcador detectado, el cual es el origen del sistema de coordenadas del marcador.
3. El elemento `OVERLAY_SIZE_PER` permite establecer una escala entre el cuadrado a superponer y el tamaño del marcador, este se define al principio del código fuente. El valor predefinido de 1 significa que el cuadrado tendrá la misma dimensión que el marcador, un valor mayor indica que el cuadrado será de mayor tamaño que el marcador.
4. Una vez definidos los puntos deseados, procedemos a calcular estos valores en relación a las coordenadas de la imagen de entrada, lo cual se realiza mediante la siguiente función la cual recibe los puntos deseados, los vectores de rotación y traslación y los parámetros de calibración de la cámara para un correcto procedimiento.

```
# Projectar los puntos
projected_desired_points, jac = cv2.projectPoints(desired_points,
```

```
rvecs,
tvecs,
cameraMatrix,
distCoeffs)
```

5. Como último paso se envía la imagen y los puntos proyectados al método para dibujarlos en la imagen de salida, esta función fue creada al inicio del código para cumplir esta objetivo.

```
# Enviar los puntos para que se dibujen
draw_points(frame, projected_desired_points)
```

6. Contenido del método draw_points()

```
def draw_points(img, pts):
    """ Dibujar los puntos en la imagen"""

    pts = np.int32(pts).reshape(-1, 2)

    img = cv2.drawContours(img, [pts], -1, (255, 255, 0), -3)

    for p in pts:
        cv2.circle(img, (p[0], p[1]), 5, (255, 0, 255), -1)

    return img
```

7. Para ejecutar se debe escribir el siguiente comando en la consola

```
python 05_aruco_detect_markers_square.py
```

8. Luego de su ejecución nuestra imagen se verá de la siguiente manera (figura 7):

2.6. Realidad Aumentada con Marcadores 2

En esta sección se utilizará una versión modificada del código anterior para detectar los marcadores y en esta ocasión superponer una imagen al marcador detectado en la imagen de entrada. Se utilizará el archivo 06_aruco_detect_markers_augmented_reality.py.

1. La imagen a superponer se carga al inicio del programa con el siguiente código y se puede ver en la figura 8.

```
# Cargar Imagen
overlay = cv2.imread("tree_overlay.png")
```

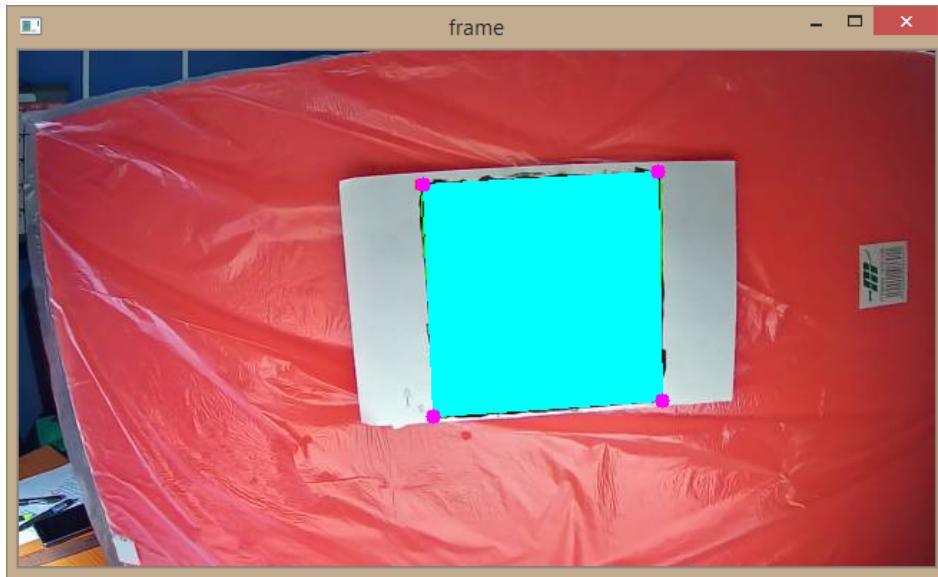


Figura 7: Cuadro superpuesto al marcador.

- Para superponer una imagen a nuestra imagen de entrada, se ha creado el siguiente método `draw_augmented_overlay`, el cual recibe los puntos deseados, la imagen a superponer y la imagen de entrada.

```
def draw_augmented_overlay(pts_1, overlay_image, image):
    """Superponer la imagen 'overlay_image' en la imagen 'image'"""

    # Defino la cuadricula de la imagen a superponer :
    pts_2 = np.float32([[0, 0],
                        [overlay_image.shape[1], 0],
                        [overlay_image.shape[1], overlay_image.shape[0]],
                        [0, overlay_image.shape[0]]]
                       )

    # Dibujar un borde para apreciar los límites de la imagen:
    cv2.rectangle(overlay_image,
                  (0, 0),
                  (overlay_image.shape[1], overlay_image.shape[0]),
                  (255, 255, 0), 10)

    # Calcular la matriz de transformación entre los puntos:
    M = cv2.getPerspectiveTransform(pts_2, pts_1)

    # Transformar la imagen usando la matriz de transformación M:
    dst_image = cv2.warpPerspective(overlay_image, M, (image.shape[1], image.shape[0]))
```



Figura 8: Imagen para superponer en el marcador.

```
# Crear la máscara:  
dst_image_gray = cv2.cvtColor(dst_image, cv2.COLOR_BGR2GRAY)  
ret, mask = cv2.threshold(dst_image_gray, 0, 255, cv2.THRESH_BINARY_INV)  
  
# Calcular una operación lógica bitwise_and usando la máscara  
image_masked = cv2.bitwise_and(image, image, mask=mask)  
  
# Sumar las 2 imágenes para crear la imagen resultado:  
result = cv2.add(dst_image, image_masked)  
return result
```

-
3. Cada sección de código se explica a continuación:
 4. En este fragmento se determina la transformación necesaria para que cada esquina de la imagen overlay_image en pts_2 coincida con los puntos del marcador identificado y almacenado en pts_1

```
# Calcular la matriz de transformación entre los puntos:  
M = cv2.getPerspectiveTransform(pts_2, pts_1)
```

5. En este caso se aplica la matriz de transformación a la imagen `overlay_image`, se redefine su tamaño a un tamaño igual al de la imagen de entrada. Esta imagen tendrá todos los píxeles en negro(0), salvo donde se ubica la imagen `overlay_image` luego de aplicar la transformación. Ver Figura 10 imágenes izquierda superior.

```
# Transformar la imagen usando la matriz de transformación M:
dst_image = cv2.warpPerspective(overlay_image, M, (image.shape[1], image.shape[0]))
```

6. Se procede con la creación de la máscara. Como primer paso se procede a crear una copia de la imagen `overlay_image` transformada pero en escala de grises. Utilizando la función de `Threshold`³ de OpenCV y usando el método binario inverso⁴, se cambia a un valor de 0 solamente los píxeles con un valor mayor a 0. Los píxeles que no cumplen (son 0 o negro) se colocan en 255 (Blanco) y en este caso se corresponden con la ubicación de la `overlay_image`, ya que este es el espacio que necesitamos que se indique en nuestra máscara. Ver Figura 9.

```
# Crear la máscara:
dst_image_gray = cv2.cvtColor(dst_image, cv2.COLOR_BGR2GRAY)
ret, mask = cv2.threshold(dst_image_gray, 0, 255, cv2.THRESH_BINARY_INV)
```

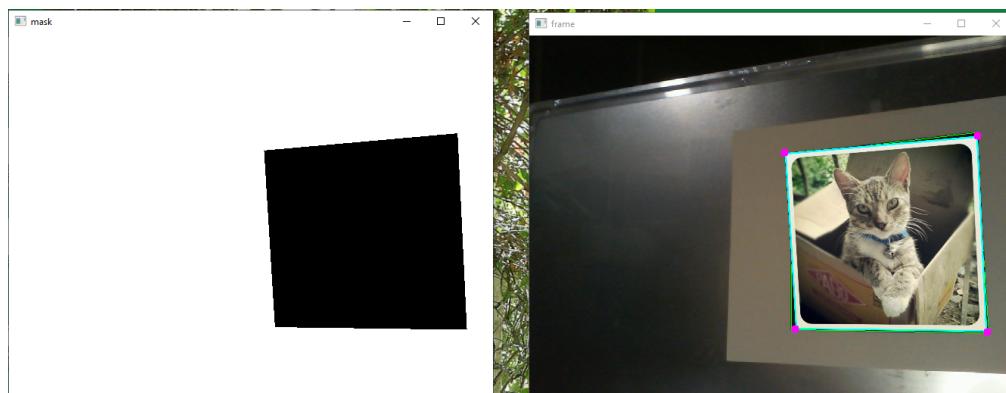


Figura 9: Resultados de la aplicación del *thresholding*.

7. En este caso nuestra máscara tiene valores 0 o negro donde estará ubicada la `overlay_image` y 255 o blanco donde no había datos, cuando se aplicó la transformación de perspectiva a la `overlay_image` los píxeles agregados se colocaron en negro. Se aplica entonces una operación lógica `bitwise_and`⁵⁶ en conjunto con la máscara. Esta operación devolverá la

³Threshold en OpenCV

⁴THRESH_BINARY_INV

⁵Operación Bitwise en OpenCV

⁶Operaciones Lógicas OpenCV

imagen de entrada original, pero con los píxeles en negro donde se ubicará `overlay_image`. Esta operación funciona como un *and* lógico y solo retorna Blanco, cuando la máscara y la imagen tienen píxeles blancos en la misma posición. Cuando la máscara tiene un valor blanco entonces este no afecta los valores originales de la imagen y esta conserva su valor. Ver Figura 10 imagen derecha superior.

```
# Calcular una operación lógica bitwise_and usando la máscara
image_masked = cv2.bitwise_and(image, image, mask=mask)
```

8. En esta operación se suman⁷ los píxeles de 2 imágenes. Los píxeles en negro (0) en una imagen, no modifican el valor del píxel en esa posición en la otra imagen. En este caso `dst_image` tiene valores en 0 donde no esta `overlay_image` mientras que `image_masked` tiene valores en 0 donde estará ubicada la `overlay_image` por lo cual el resultado es la unión de ambas imágenes. Ver Figura 10 imagen inferior.

```
# Sumar las 2 imágenes para crear la imagen resultado:
result = cv2.add(dst_image, image_masked)
```

9. Las salidas parciales de este código se pueden apreciar en la figura 10.

10. Para ejecutar se debe escribir el siguiente comando en la consola

```
python 06_aruco_detect_markers_augmented_reality.py
```

11. Luego de su ejecución nuestra imagen se verá de la siguiente manera (figura 11):

⁷Suma OpenCV

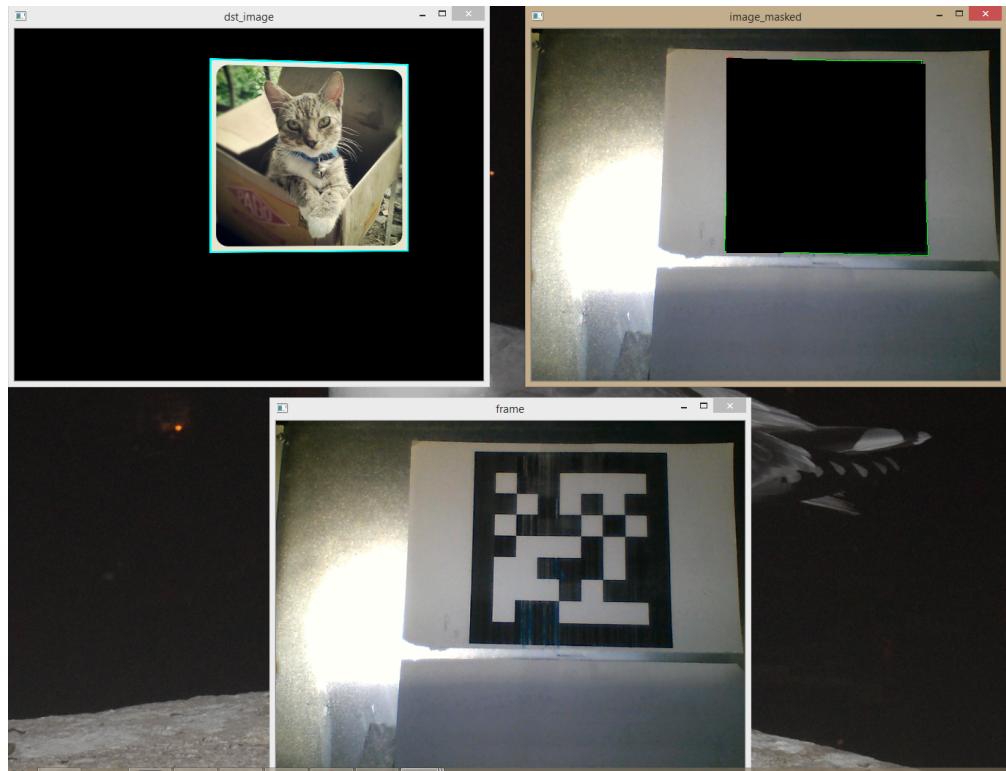


Figura 10: Diferentes resultados que se obtienen al aplicar los diversos pasos de la función.

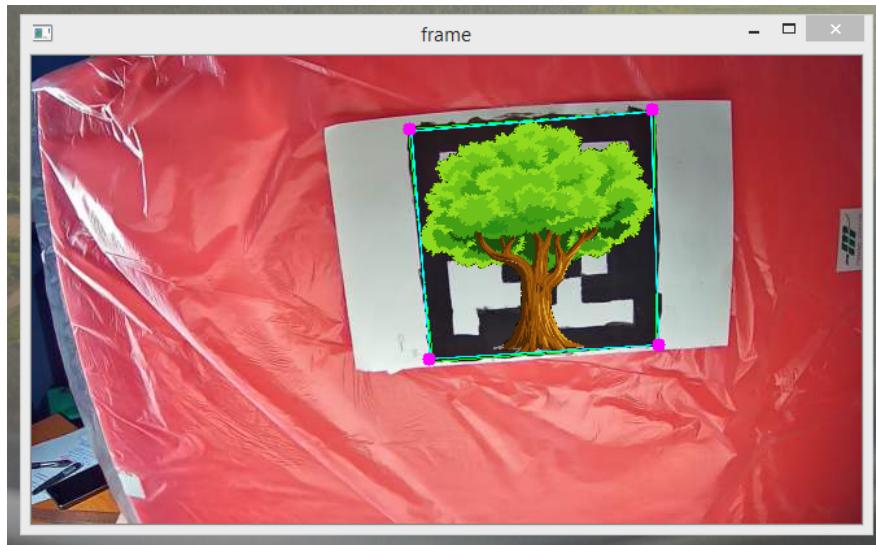


Figura 11: Imagen superpuesta al marcador.

Entregables para la Sección

Realizar las siguientes operaciones:

1. Utilizando el código `06_aruco_detect_markers_augmented_reality.py` solo podrá identificar un marcador a la vez, al tener más marcadores en la imagen se producirá un error, modifique este código para que se puedan identificar múltiples marcadores en una imagen en tiempo real. Guarde la imagen resultante de la ejecución de este código y agréguela a su informe.
2. Utilizando el código anterior modificado, edite el mismo y configúrelo para que solo se reemplace cuando el marcador identificado es el número 73 del diccionario. Guarde la imagen resultante de la ejecución de este código y agréguela a su informe.

Referencias

- [1] A. Fernández Villán, *Mastering OpenCV 4 with Python: a practical guide covering topics from image processing, augmented reality to deep learning with OpenCV 4 and Python 3.7. Mastering Open Source Computer Vision four with Python*. Birmingham: Packt Publishing, 2019. [Online]. Available: <https://cds.cern.ch/record/2674578>
- [2] E. Cruz, S. Orts-Escalano, F. Gomez-Donoso, C. Rizo, J. C. Rangel, H. Mora, and M. Cazorla, "An augmented reality application for improving shopping experience in large retail stores," *Virtual Reality*, vol. 23, no. 3, pp. 281–291, 2019.
- [3] C. Ricolfe Viala and A. J. Sánchez Salmerón, "Procedimiento completo para el calibrado de cámaras utilizando una plantilla plana," *Revista Iberoamericana de Automática e Informática Industrial RIAI*, vol. 5, no. 1, pp. 93 – 101, 2008. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S1697791208701262>