L3 - *« Computer Vision »* *L. Mascarilla,*	TP n°4 - **Réalité augmentée** **Aruco**	

L'objectif de ce TP est d'expérimenter mettre en œuvre les principes de réalité augmentée vus en cours. Nous allons insérer des objets simples dans la scène captée par la caméra JeVois. La méthode s'appuie sur les marqueurs Aruco.

La JeVois souffre d'un problème avec certaines fonctions (elle a un GROS bug!). Nous allons donc travailler en 2 temps :

- Développement sur votre PC en utilisant la JeVois comme une WebCam (pensez à modifier le fichier videomapping.cfg comme indiqué dans le TP2). Et en utilisant comme base le fichier skel_OpenCV.py du TP3. Je vous suggère d'utiliser une résolution de 640x480.

- Portage du code sur la JeVois

Partie 1 : Marqueurs Aruco sur Ordinateur portable

La documentation de la bibliothèque aruco pour OpenCV est complète en C++ (mais encore très partielle en python!).

Vous trouverez cette documentation sur cette page :

https://docs.opencv.org/4.9.0/d9/d6a/group__aruco.html

ainsi qu'un tutorial :

https://docs.opencv.org/4.9.0/d5/dae/tutorial_aruco_detection.html

Vous pouvez vous y référer et « adapter » le code C++ en python.

La page de référence de ces marqueurs est celle-ci :

https://www.uco.es/investiga/grupos/ava/portfolio/aruco/

1) **Initialisation, détection et affichage des marqueurs :**

Avant toutes choses, vous devez inclure la bibliothèque aruco :

```
import cv2.aruco as aruco
```

Puis vous devrez choisir le dictionnaire (i.e. la liste des marqueurs) utilisé dans votre application. Il est possible d'en générer un adapté à vos besoins :

https://docs.opencv.org/4.9.0/d5/dae/tutorial_aruco_detection.html ou
http://chev.me/arucogen/ pour une génération en ligne.

Pour l'instant, vous utiliserez celui-ci : aruco.DICT_4X4_50. Un échantillon de ces marqueurs est visible ici :
http://jevois.org/moddoc/DemoArUco/screenshot2.png
Vous pourrez les utiliser soit sous forme imprimée (demandez à l'enseignant) ou sur un écran de PC ou de portable.

Il est possible de les récupérer dans une variable par l'instruction :

```
dico = aruco.getPredefinedDictionary(aruco.DICT_4X4_50)
```

Une fois ceci réalisé, vous pourrez récupérer les paramètres par défaut (seuils, taille de la fenêtre pour le seuillage adaptatif…) de la méthode de détection par :

```
parameters=aruco.DetectorParameters()
```

Comme nous l'avons vu en cours, la méthode utilise une image en niveau de gris.
Pour **Opencv < 4.9**, la méthode de détection des marqueurs est aruco.detectMarkers, elle retourne les coins des marqueurs détecteurs, les identifiants (numéros) et les détections rejetées :

```
corners, ids, rejectedImgPos = aruco.detectMarkers(
   inimg, aruco_dict, parameters=parameters
)
```

Si vous utilisez une version d'**OpenCV ≥ 4.9**, il utiliser :

```
detector = aruco.ArucoDetector(dico,parameters)
```

et

```
corners, ids, rejectedImgPos = detector.detectMarkers(inimg)
```

Une fois cette détection effectuée, la fonction suivante trace ces marqueurs sur l'image :

```
outimg = aruco.drawDetectedMarkers(inimgrgb, corners, ids)
```

Mettez en œuvre cette chaîne de traitement pour faire afficher sur l'image (couleur) de la caméra les marqueurs détectés ainsi que les marqueurs rejetés. Ces derniers seront affichés en rouge.

2) Affichage des axes

Pour effectuer l'affichage d'objets sur la scène vous devez connaître les paramètres **intrinsèques** et **extrinsèques** du dispositif d'acquisition.

a) Les paramètres **intrinsèques**, et les coefficients de **distorsion**, peuvent être récupérés par la fonction suivante qui nécessite la **largeur** et la **hauteur** de l'image affichée par la caméra :

```
def loadCameraCalibration(w, h):
   cpf = f"/jevois/share/camera/calibration{w}x{h}.yaml"
   fs = cv2.FileStorage(cpf, cv2.FILE_STORAGE_READ)
   if fs.isOpened():
      camMatrix = fs.getNode("camera_matrix").mat()
```

```
        distCoeffs = fs.getNode("distortion_coefficients").mat()
        jevois.LINFO(f"Loaded camera calibration from {cpf}"
    else:
        jevois.LFATAL(f"Failed to read camera parameters from file [{cpf}]"
    return camMatrix, distCoeffs
```

Remarque : la variable cpf contient le chemin vers le fichier de calibration de la JeVois (d'où le /jevois/…). Sur votre PC vous utiliserz le fichier calibration640x480.yaml disponible sur Moodle.

Affichez ces paramètres ci-dessus. **Interprétez-les.**

b) Les paramètres **extrinsèques** sont récupérés par la fonction suivante pour **OpenCV < 4.9**:

```
rotation, translation, _ = aruco.estimatePoseSingleMarkers(
    corners, taille_marqueur_m, camMatrix, distCoeffs
)
```

taille_marqueur_m est la largeur réelle en mètre du marqueur.

Pour pouvoir utiliser cette fonction, il faut donc qu'au moins un marqueur **(identifiés par corners et ids)** ait été repéré.

Vous remarquerez également que chaque marqueur renvoie une matrice de rotation et un vecteur de translation (paramètres extrinsèques).

Chaque axe peut être tracé par la fonction :

```
cv2.drawFrameAxes(image, camMatrix, distCoeffs, rotation, translation, longueur)
```

où longueur est la longueur en mètre des axes.

Pour **OpenCV ≥ 4.9**, estimatePoseSingleMarkers n'existe plus, il faut utiliser la fonction générique solvePNP. Pour simplifier son utilisation vous pouvez utiliser la fonction suivante qui s'utilise avec les mêmes paramètres :

```
def my_estimatePoseSingleMarkers(corners, marker_size, mtx, distortion):
    '''
            https://stackoverflow.com/questions/75750177/solve-pnp-or-estimate-pose-single-markers-which-is-better
    This will estimate the rvec and tvec for each of the marker corners detected by:
       corners, ids, rejectedImgPoints = detector.detectMarkers(image)
    corners - is an array of detected corners for each detected marker in the image
    marker_size - is the size of the detected markers
    mtx - is the camera matrix
    distortion - is the camera distortion matrix
    RETURN list of rvecs, tvecs, and trash (so that it corresponds to the old estimatePoseSingleMarkers())
    '''
    marker_points = np.array([[-marker_size / 2, marker_size / 2, 0],
```

```
                    [marker_size / 2, marker_size / 2, 0],
                    [marker_size / 2, -marker_size / 2, 0],
                    [-marker_size / 2, -marker_size / 2, 0]], dtype=np.float32)
  trash = []
  rvecs = []
  tvecs = []

  for c in corners:
                    nada, R, t = cv2.solvePnP(marker_points, c, mtx, distortion, False,
cv2.SOLVEPNP_IPPE_SQUARE)
      rvecs.append(R)
      tvecs.append(t)
      trash.append(nada)
  return rvecs, tvecs, trash
```

3) Affichage d'objets 3D.

Comme nous disposons de tous les paramètres de la caméra, il est « facile » d'insérer un objet dans la scène.

Pour cela, il suffit de disposer des coordonnées dans le monde réel des points que nous voulons insérer dans l'image et d'y appliquer la transformation vue en cours : multiplication par la matrice extrinsèque (rotation|translation) des coordonnées puis par la matrice intrinsèque et enfin par les paramètres de distorsion. Tout cela est réalisé par la fonction :

```
imagePoints, _ = cv2.projectPoints(
  coordonnees, rotation, translation, camMatrix, distCoeffs
)
```

où coordonnees **contient les coordonnées (exprimées en m) des points. Chaque point est un tuple de trois valeurs** (x,y,z)**. L'ensemble des points est regroupé dans une liste. Ces coordonnées doivent être exprimées relativement au centre (0,0,0) du repère de chaque marqueur.**

Le retour de la fonction est une liste (pensez à la convertir en numpy array de type int**) qui contient les coordonnées des points 2D (sur l'image) obtenues grâce à la projection des points 3D (scène réelle) contenus dans** coordonnees**. Par exemple, si vous avez 8 points (comme les sommets d'un cube), les coordonnées projetées du 1er point sont obtenus par :**

```
imagePoints[0].ravel()
```

et de façon générale par imagePoints[i].ravel()**, si i varie dans [0,7].**

Il suffit ensuite de les afficher sur l'image (frame) par les fonctions standard de tracé d'OpenCV : https://docs.opencv.org/4.9.0/dc/da5/tutorial_py_drawing_functions.html

Classiquement, on trace les segments reliant les points deux à deux.

Tracez un objet géométrique simple comme un cube (facile !) ou une pyramide.

4) Incrustation d'images

Cette partie pose problème quand elle est effectuée sur la JeVois : Le module créé n'est utilisable que jusqu'à ce que la JeVois soit déconnectée ! Une fois

reconnectée, le module ne fonctionne plus !

En utilisant Python, la JeVois ne permet pas d'utiliser OpenGL (il est toutefois possible de le faire en C++). Le placage de texture et les tracés rapides en 3D sont donc limités. Toutefois, nous pouvons incruster facilement des images avec transparence dans le flux de la caméra en utilisant les possibilités de mélange d'images (blending et overlay) d'OpenCV.

Vous trouverez un exemple d'image carrée avec un fond transparent sur Moodle (mais vous pouvez bien sur prendre celle qui vous convient).

L'idée est ici de superposer votre image après projection à l'image de la caméra sur chacune des faces supérieures des cubes placés sur les marqueurs. On utilisera cv2.addWeighted.
Afin d'effectuer la projection, vous devez disposer de la matrice de projection homographique. Elle peut être calculée à partir des matrices intrinsèques et extrinsèques. Il est également possible de la calculer à partir des coordonnées de 4 points appariés (*i.e.* pour chaque point de l'image à projeter il faut savoir en quel point de la scène il est transformé).

Pour cela il faut procéder en 5 temps (!!):

1. Obtenir les coordonnées des points dans le plan image de la face sur laquelle vous voulez effectuer la projection. Vous les avez obtenus à l'étape précédente après cv2.projectPoints.

2. Obtenir les coordonnées des 4 coins de l'image à projeter (facile : elles sont calculables directement à partir de la largeur et hauteur de l'image à projeter)

3. Estimer la matrice homographique par la fonction cv2.getPerspectiveTransform

4. Effectuer la projection de l'image dans la scène par cv2.warpPerspective qui utilise la matrice de projection homographique. Vous obtenez alors une image qui contient l'image déformée.

5. Superposez cette image déformée à l'image de la caméra en utilisant cv2.addWeighted

Remarque : pour éviter d'avoir à chercher l'ordre d'appariement, vous pouvez utiliser la fonction suivante qui trie les coordonnées de gauche à droite et de haut en bas dans une image. Il suffit d 'appliquer cette fonction aux deux ensembles de points à apparier avant de faire l'estimation de l'homographie.

```
def order_points(pts):
   # d'après https://www.pyimagesearch.com/2016/03/21/ordering-coordinates-clockwise-with-python-and-opencv/
   def distance(a, b):
      d = ((a[0] - b[0]) ** 2 + (a[1] - b[1]) ** 2) ** 0.5
      return d
```

```
    xSorted = pts[np.argsort(pts[:, 0]), :]

    leftMost = xSorted[:2, :]
    rightMost = xSorted[2:, :]


    leftMost = leftMost[np.argsort(leftMost[:, 1]), :]
    (tl, bl) = leftMost

    D=[distance(tl[np.newaxis][0],rightMost[0,:]),distance(tl[np.newaxis][0],rightMost[1,:])]
    (br, tr) = rightMost[np.argsort(D)[::-1], :]


    return np.array([tl, tr, br, bl], dtype="float32")
```

Partie 2 : Marqueurs Aruco sur JeVois

Portez le code sur la JeVois : attention à ne pas la déconnecter (!) et à adapter les chemins vers les fichiers de la JeVois :

- "/jevois/data/GreenEye.png"
- /jevois/share/camera/calibration{}x{}.yaml