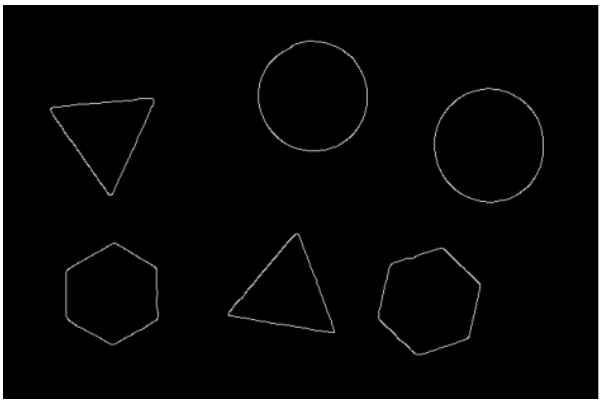
Institut d'Optique Graduate School
INTERFAÇAGE NUMÉRIQUE

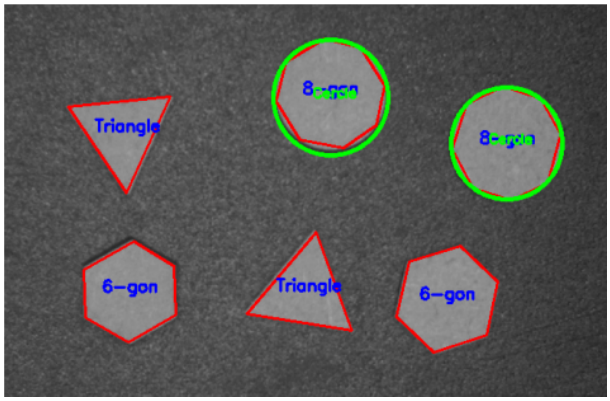
Travaux Pratiques

Semestre 6

Vision Industrielle

Objectifs des 4 séances
TP 3-4 / Détection d'objets



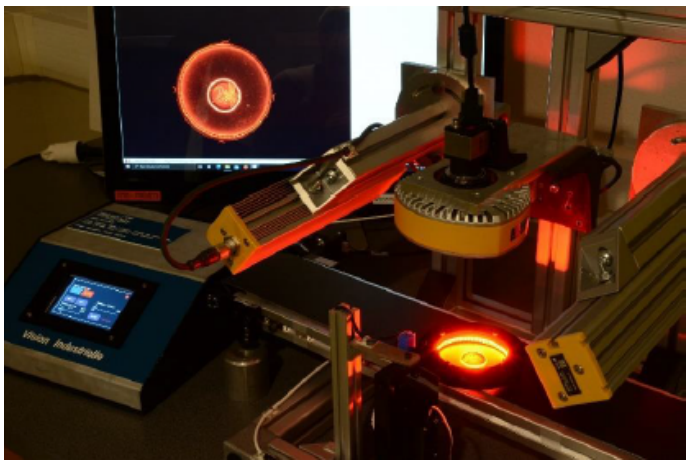


Ce sujet est disponible au format électronique sur le site du LEnsE - https ://lense.institutoptique.fr/ dans la rubrique Année / Première Année / Interfaçage Numérique S6 / Bloc 2 Vision Industrielle.

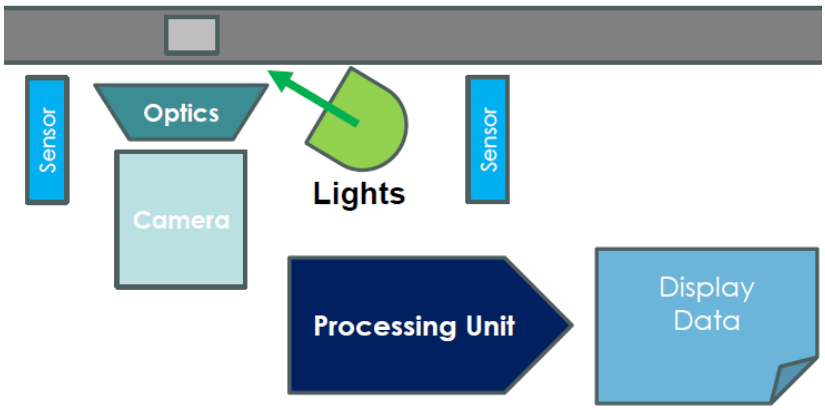
Ce sujet a été co-conçu par une équipe d'étudiant·es - Joséphine BECHU, Justine GABRIEL et Paul CHENEAU - lors d'un projet DEPhI de 2A - 2025-2026 - et l'équipe pédagogique d'Interfaçage Numérique.

Vision Industrielle

La **vision industrielle** est une technologie qui permet à des machines d'**analyser automatiquement des scènes** pour **contrôler**, **guider** ou **inspecter** des objets sur des processus de production. Elle repose sur l'utilisation de **caméras**, d'**optique**, d'**éclairages** spécifiques (ou contraints), de **capteurs** et d'algorithmes de **traitement d'image**.



Elle a pour but de **prendre des décisions automatiques** (ou aider l'être humain dans sa prise de décision) vis-à-vis d'un (ou plusieurs) objet(s) dans une scène spécifique : détecter des défauts ou des irrégularités, compter ou trier..., en rejetant ou validant automatiquement des produits, tout en assurant une constance de la qualité et de la répétabilité des opérations.



Acquis d'Apprentissages Visés (AAV)

À la fin de cette série de 4 séances, les étudiant·es seront capable de :

— Décomposer et paramétrer une chaîne de vision industrielle complète (du capteur au traitement de l'image),
— Comprendre les compromis physiques et numériques de chaque maillon,
— Réaliser un prototype d'inspection ou de mesure simple (tri d'objets),
— Justifier leurs choix de configuration (résolution, focale, éclairage...)

Ressources

Un **kit d'images** est disponible sur le site du LEnsE dans la rubrique *Année / Première Année / Interfaçage Numérique S6 / Bloc 2 Vision Industrielle / Kit d'images*.

Des codes en Python, proposant des exemples à tester, sont disponibles sur le site du LEnsE dans la rubrique *Année / Première Année / Interfaçage Numérique S6 / Bloc 2 Vision Industrielle / Répertoire vers codes à tester*.

Un fichier archivé, nommé _STEP_BY_STEP.ZIP*, regroupe l'ensemble des codes à tester au cours de cette séance, ainsi que les images à traiter.*

Déroulement

Les sujets **TP1** et **TP2** se font en binôme et sont interchangeables (4 binômes commenceront par la TP1 lors de la première séance et les 4 autres binômes commenceront par le TP2).

Les sujets **TP3** et **TP4** se font par groupe de 4 étudiant·es.

TP1 - Banc de vision industrielle

Le **TP1** se fera sur un banc de vision industrielle simple incluant une caméra, un objectif (focale : 5 / 8 / 10 mm), un éclairage Effilux Ring-RGB et un ordinateur.

Voir le sujet de la séance TP1. Le sujet est disponible au format électronique sur le site du LEnsE - https ://lense.instituto
dans la rubrique Année / Première Année / Interfaçage Numérique S6 / Bloc 2 Vision Industrielle.

TP2 - Manipulations de base sous OpenCV

Le **TP2** se fera sur un ordinateur possédant une installation de Python (3.11 ou 3.13) et des bibliothèques standard, dont OpenCV.

Voir le sujet de la séance TP2. Le sujet est disponible au format électronique sur le site du LEnsE - https ://lense.instituto
dans la rubrique Année / Première Année / Interfaçage Numérique S6 / Bloc 2 Vision Industrielle.

TP3-4 - Détection d'objets

Le détail des séances 3 et 4 est donné dans la suite de ce document.

Accumulation de preuves / Méthode de travail

Il est conseillé pour ces TP de **créer un nouveau projet PyCharm** sur votre session (*attention à l'endroit où vous stockerez ce projet - U : sur les sessions Windows de l'IOGS*).

Les différents algorithmes que vous serez amenés à modifier ou créer, pourront vous resservir dans d'autres projets. Nous vous conseillons donc fortement de les **sauvegarder** précieusement et de les **commenter** autant que possible afin de retrouver rapidement les principes mis en jeu derrière les fonctionnalités de OpenCV.

Il serait également pertinent de votre part de rédiger un **journal de bord** sur cette série de TP en incluant les résultats (images, histogrammes...) et vos analyses des fonctionnalités et de leur intérêt en traitement d'images.

Objectif des séances 3 et 4

TP3 - Manipulations avancées sous OpenCV / Limites du banc de vision

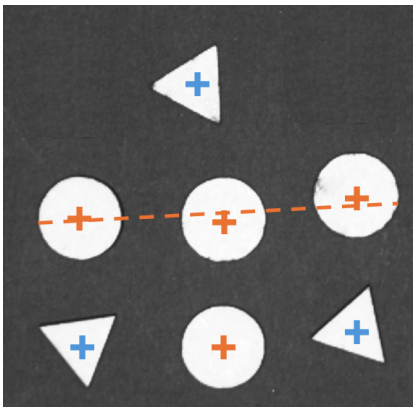
Sous OpenCV :
— Détecter des formes [40 min]
— Détecter des couleurs [40 min]
— Changer d'espace de couleur [40 min]
Segmentation de l'image ?
Sur le banc de VI :
— Calibration de la chaîne sur les cubes de couleurs (couleur) [40 min]
— Calibration de la chaîne sur des objets colorés et des objets de forme déterminée[90 min]
Le sujet du TP3 est dans la suite de ce document.

TP4 - Détection d'objets - Contrôle qualité

L'objectif final de cette série de TP est de détecter le nombre d'objets d'une forme ou d'une couleur particulière à partir d'une série d'images (1 en RGB ou 3 en niveau de gris sous éclairage particulier).

On se basera sur la règle du **jeu du Morpion** (ou *Tic Tac Toe* en anglais) qui oppose deux joueurs qui doivent aligner 3 marques (de même forme ou couleur) sur une grille de 3 x 3 cases.

Premiers essais sur une grille de jeu TicTacToe (morpion) :



Cette mission "ludique" est cependant très proche des **tâches d'inspection en industrie**, afin par exemple de valider si les pièces requises pour un assemblage sont bien présentes (simuler le packaging d'un produit fini - 6 pièces de forme et de couleurs distinctes par exemple).

On pourra ajouter également des mesures de surface des pièces, de colorimétrie pour valider la couleur des dépôts faits sur les pièces... pour vérifier leur **conformité à un cahier des charges**.

Il est également crucial pour les chaînes de production, par exemple, d'assurer une **cadence de fabrication et de validation importante**. Les algorithmes de détection doivent donc pouvoir s'exécuter rapidement et la vitesse de défilement des pièces sous le système imageant peut devenir limitant.

Il est enfin nécessaire de valider ces systèmes de prise de décision en mesurant leur **taux d'erreur** vis-à-vis des grandeurs à tester. Ce taux d'erreur doit être proche de 0, une tolérance peut être acceptée et peut être différente selon le type de défauts à identifier, le type de matériaux utilisés...

OpenCV - Détection de formes et de couleurs

Une partie des algorithmes de traitement sont donnés. Il est fortement conseillé de **réaliser une analyse poussée** de chacune des étapes de ces algorithmes afin de comprendre l'intérêt de l'utilisation de ces techniques de base.

Vous pouvez par exemple **décrire ces algorithmes** par l'intermédiaire d'un graphique (algorigramme) et **afficher les résultats intermédiaires** (et en faire une capture d'écran) tout en **commentant les résultats**.

CV-A - Détecter des formes [40 min]

Détection de contours sur une image "propre"

➜ **M** Ouvrir le fichier 20_DETECTION_FORME.PY du répertoire des codes à tester.

➜ **M** Exécuter ce code.

➜ **Q** Que fait ce programme ? A quoi correspondent les images affichées ? A quoi correspondent les données affichées ?

➜ **M** Modifier le paramètre MIN_AREA (100, 1000, 10000).

➜ **Q** A quoi sert ce paramètre ? Quelle est son unité ? Comment la transformer en unités plus conventionnelles ?

➜ **M** Modifier le paramètre APPROX_FACTOR (0.01, 0.05, 0.1).

➜ **Q** A quoi sert ce paramètre ?

➜ **Q** A quoi correspondent les variables `cX`, `cY`, `area` et `peri` ?

Détection de cercles

➜ **M** Ouvrir le fichier 21_DETECTION_FORME_CERCLES.PY du répertoire des codes à tester.

➜ **M** Exécuter ce code.

➜ **Q** Quelle amélioration apporte ce programme ?

Détection de contours sur une image "réelle"

L'image précédente était "propre" (sans bruits dûs au capteur par exemple). Nous allons nous intéresser à présent à une image provenant d'une chaine d'acquisition.

➜ **M** A partir du code précédent, ouvrir l'image FORMES_BLANC_30MS.PNG du répertoire du kit d'images.

➜ **Q** L'algorithme est-il efficace ?

➜ **M** Ouvrir le fichier 22_DETECTION_FORME_PLUS.PY du répertoire des codes à tester.

➜ **M** Exécuter ce code.

➜ **Q** Quelle étape est-elle ajoutée par rapport à la version précédente du code ?

➜ **M** Ajouter un traitement morphologique de type *opening* (ouverture) sur l'image binarisée (noyau carré de 3x3, puis 5x5).

➜ **M** Ajouter l'affichage de l'image ainsi traitée.

➜ **Q** Que permet cette étape dans l'algorithme de détection ?

➜ **Q** L'algorithme de détection de cercles fonctionne-t-il avec cette étape supplémentaire ?

Détection de cercles sur une image "réelle"

L'**algorithme de Hough** qui permet la recherche de cercles dans une image se base sur des images en nuance de gris (et non binarisée). Il s'appuie sur des gradients d'intensité et des bords continus et doux pour détecter les transitions (contours des cercles).

Il n'est donc pas possible d'utiliser une opération morphologique d'ouverture sur l'image.

➜ **M** Ouvrir le fichier 23_DETECTION_FORME_BLUR_CERCLES.PY du répertoire des codes à tester.

➜ **M** Exécuter ce code. Vérifier que l'algorithme n'est pas efficace.

➜ **M** Ajouter un filtrage de type *blur* gaussien (*cv2.GaussianBlur*) sur l'image grise (noyau carré de 7x7 et $\sigma = 1.9$).

➜ **M** Ajouter l'affichage de l'image ainsi traitée.

➜ **Q** Que permet cette étape dans l'algorithme de détection ?

➜ **M** Modifier la taille du noyau (3x3, 5x5, 15x15) et de la valeur de σ (1.9, 3, 5, 10).

➜ **Q** Quel est l'impact du choix de ces deux paramètres ?

CV-B - Détecter des couleurs [40 min]

Jouer avec les canaux de couleurs

➜ **M** Ouvrir et afficher l'image *couleurs_4.png* du kit d'images fourni, au format RGB.

➜ **M** Créer une copie de la matrice image (fonction *copy()* de Numpy).

➜ **M** Forcer à 0 tous les pixels du canal bleu de la copie de l'image et afficher la nouvelle image.

Utiliser l'espace colorimétrique HSV

L'espace colorimétrique **HSV** (*Hue, Saturation, Value*) représente les couleurs de manière intuitive en les séparant selon :

— la **teinte** (Hue), qui correspond au type de couleur et est généralement exprimée comme un angle sur un cercle (rouge, vert, bleu, etc.),
— la **saturation** (Saturation), qui mesure la pureté ou l'intensité de la couleur (d'une couleur vive à un gris),
— la **valeur** (Value), qui représente la luminosité globale.

Contrairement à l'espace RVB, HSV est conçu pour se rapprocher de la perception humaine, ce qui le rend particulièrement adapté aux tâches de vision par ordinateur, notamment la détection et la segmentation de couleurs, car il permet de dissocier plus facilement l'information de couleur des variations d'éclairage.

➜ **M** Ouvrir le fichier 24_COULEUR_HSV.PY du répertoire des codes à tester.

➜ **M** Exécuter ce code.

➜ **Q** Que représentent les deux lignes de figure affichées par ce programme ?

➜ **Q** Comment pouvez-vous interpréter les deux histogrammes ?

➜ **M** Ajouter un filtrage de type *blur* gaussien (*cv2.GaussianBlur*) sur l'image rgb FORMES_BRUIT.PNG (noyau carré de 7x7 et $\sigma = 1.9$).

➜ **Q** Quel est l'effet sur l'histogramme associé à cette image ? Quel serait l'inconvénient de cette technique, par rapport à la détection de couleur via l'espace HSV ?

Segmenter une image en fonction de la couleur

➜ **M** Ouvrir le fichier 24B_HSV_RGB.PY du répertoire des codes à tester.

➜ **M** Exécuter le code.

➜ **Q** Expliquer le principe utilisé dans chacune des segmentations, en RVB et HSV. On s'intéressera en particulier aux bornes utilisées.

➜ **Q** Quelle méthode est la plus efficace pour détecter la couleur rouge ?

➜ **M** Ajouter la détection de la zone bleue, de l'image initialement ouverte, par la méthode qui vous parait la plus pertinente. Afficher une nouvelle image avec le résultat final.

CV-C - Passer de la couleur à des niveaux de gris [40 min]

Il existe plusieurs méthodes pour transformer une image en RVB vers une image en nuance de gris par exemple.

Moyenne des canaux R,V,B

Cette méthode est la plus simple. Chaque pixel de l'image en gris est la moyenne des pixels des canaux rouge, vert et bleu de l'image en couleur :

$$Pixel_{Gray} = \frac{Pixel_R + Pixel_V + Pixel_B}{3}$$

Pondération en fonction de la perception humaine

Cette méthode est une moyenne pondérée des valeurs des pixels R, V, B de l'image couleur :

$$Pixel_{Gray} = 0.299 \cdot Pixel_R + 0.587 \cdot Pixel_V + 0.114 \cdot Pixel_B$$

Les coefficients de cette méthode proviennent de la sensibilité relative de l'œil humain aux différentes couleurs et ont été standardisés à l'origine pour la télévision analogique (NTSC). Ils sont aujourd'hui utilisés comme une approximation fidèle de la perception visuelle de la luminosité.

La méthode de conversion fournie par la bibliothèque OpenCV se base sur cette pondération.

➜ **M** Ouvrir le fichier 30_COLORSPACE.PY du répertoire des codes à tester.

➜ **Q** Que fait ce programme ? A quoi sert l'instruction CMAP='GRAY' de la fonction *imshow* de *Matplotlib* ?

➜ **M** Créer une image en nuance de gris utilisant la méthode de la moyenne des trois canaux. *Attention aux types des données dont vous faites la moyenne...*

➜ **M** Afficher l'image résultante.

➜ **M** Comparer les images obtenues par la moyenne classique et la moyenne pondérée.

Utiliser l'espace colorimétrique YUV

L'espace colorimétrique RVB est très utilisé dans le domaine du numérique (affichage, acquisition d'images) pour sa facilité de mise en oeuvre.

Cependant, ce n'est **pas** le plus **adapté vis-à-vis de la perception humaine** où la luminance et la couleur sont séparées.

Des espaces comme YUV, YIQ, ou YCbCr séparent la composante de luminance (Y) des composantes de chrominance (U et V).

➜ **M** Ouvrir le fichier 31_COLORSPACE _YUV.PY du répertoire des codes à tester.

➜ **M** Comparer alors l'image en niveau de gris obtenue par la méthode de moyennage pondérée et le canal Y de cette conversion.

➜ **Q** Que pouvez-vous conclure sur la méthode de calcul utiliser pour la luminance (Y) ?

Il existe d'autres espaces colorimétriques dans le domaine numérique. Voici un résumé non exhaustif :

Espace colorimétrique	Avantages
RGB	Simple, utilisé pour les écrans et le rendu des couleurs.
HSV / HSL	Intuitif pour manipuler la couleur (teinte, saturation).
YUV / YCbCr	Sépare luminance et chrominance.
CIE-Lab	Uniformité perceptuelle, idéal pour mesurer les différences de couleur.
CMY(K)	Optimisé pour l'impression.
XYZ	Modèle basé sur la perception humaine.

Banc VI - Calibration sur flux vidéo

On se propose dans cette partie de tester les algorithmes de détection sur un flux vidéo, sans passer par l'interface utilisée lors d'une précédente séance.

VI-A - Tester les algorithmes sur un flux vidéo [40 min]

➜ **M** Ouvrir le fichier CAMERA_FLUX_LENSEPY_SIMPLE.PY du répertoire des codes à tester.

➜ **M** Tester le programme, en ayant au préalable connecté une caméra de type Basler. Tester en particulier l'interaction avec l'utilisateur (touches 'a', 'i', 'c', 'g').

➜ **Q** A quoi servent les options COLOR_MODE et DISPLAY_RATIO ?

➜ **Q** Faire un algorigramme (algorithme du programme en version graphique) de ce code.

➜ **M** Modifier le code pour ajouter une option (par exemple la lettre 'e') : permettre l'affichage de l'image subissant une érosion avec un noyau circulaire de 7x7. Tester votre code.

VI-B - Calibrer la chaîne d'acquisition [90 min]

Pour cette section, les binômes étant sur des paillasses disposant d'une caméra pourront utiliser directement les images provenant de la caméra.

Pour les autres binômes, vous pourrez vous reposer sur les images suivantes :

- CUBES_RGB.PNG : acquisition des 4 cubes de couleur sous un éclairage blanc (*ring* RGB Effi-Ring) en mode BayerRG8 ;
- CUBES_R.PNG : acquisition des 4 cubes de couleur sous un éclairage rouge (*ring* RGB Effi-Ring) en mode Mono8 ;
- CUBES_G.PNG : acquisition des 4 cubes de couleur sous un éclairage vert (*ring* RGB Effi-Ring) en mode Mono8 ;
- CUBES_B.PNG : acquisition des 4 cubes de couleur sous un éclairage bleu (*ring* RGB Effi-Ring) en mode Mono8 ;
- FORMS.PNG : acquisition d'une dizaine de pièces en bois de formes différentes sous un éclairage blanc (*ring* RGB Effi-Ring) en mode Mono8 ;
- FORMS_TOE.PNG : acquisition d'une dizaine de pièces en bois de formes différentes - alignées - sous un éclairage blanc (*ring* RGB Effi-Ring) en mode Mono8 ;

Ces images sont disponibles sur le site du LEnsE - https ://lense.institutoptique.fr/ dans la rubrique Année / Première Année / Interfaçage Numérique S6 / Bloc 2 Vision Industrielle / Cubes/Formes.

Détecter des cubes colorés

On se propose ici de détecter la couleur des cubes fournis (jaune, rouge, bleu et vert) de manière un peu plus automatisée et d'extraire certaines informations de ces objets (position, aire...).

➜ **Q** Proposer une méthode (algorithme) à mettre en place pour détecter la présence d'un élément d'une des 4 couleurs à détecter, ainsi que sa position.

➜ **M** A partir de l'exemple CAMERA_FLUX_LENSEPY_SIMPLE.PY (en remplaçant si besoin l'image provenant de la caméra par l'ouverture d'une image fixe), ajouter cette fonctionnalité.

➜ **Q** Proposer une méthode (algorithme) à mettre en place pour détecter la présence de tous les éléments d'une couleur donnée et de compter le nombre d'objets présents.

➜ **M** A partir de l'exemple CAMERA_FLUX_LENSEPY_SIMPLE.PY (en remplaçant si besoin l'image provenant de la caméra par l'ouverture d'une image fixe), ajouter cette fonctionnalité.

➜ **M** A partir de l'exemple CAMERA_FLUX_LENSEPY_SIMPLE.PY (en remplaçant si besoin l'image provenant de la caméra par l'ouverture d'une image fixe), créer un programme qui permette de lister les cubes des 4 couleurs et d'afficher pour chacun d'eux : les coordonnées de leur centre, leur surface, la couleur...

Il est possible d'avoir une estimation du temps d'exécution d'un code en Python par la méthode proposée ci-dessous :

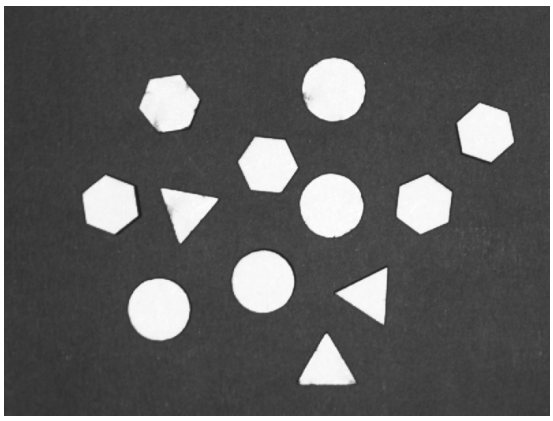
```
1 import time
2
3 start = time.perf_counter()
4 # Code to evaluate
5 end = time.perf_counter()
6
7 print(f"Total Time : {(end - start)*1000:.2f} ms")
```

➜ **M** A partir de cet exemple, mesurer le temps de traitement de détection des cubes de couleurs.

Détecter des formes particulières

On souhaite à présent s'intéresser aux pièces en bois de 3 formes différentes (cercle, triangle, hexagone).

A terme, nous aimerions pouvoir détecter si des formes d'un même type sont alignées sur une grille (principe du jeu Tic-tac-toe).



➜ **Q** A partir des exemples fournis dans la section précédente (OpenCV), proposer une méthode (algorithme) à mettre en place pour détecter les éléments en fonction de leur forme. On s'intéressera notamment aux coordonnées du centre de la pièce et à leur forme.

➜ **M** A partir de l'exemple CAMERA_FLUX_LENSEPY_SIMPLE.PY (en remplaçant si besoin l'image provenant de la caméra par l'ouverture d'une image fixe), mettre en oeuvre cet algorithme et tester votre code.

La suite de cette partie pourra être continuée lors de la séance 4. Vous pourrez notamment travailler par équipe de 4 pour améliorer la détection des formes et leur emplacement.

Pour finaliser ce mini-projet, il reste à mettre en place la détection des alignements des formes et vérifier que 3 formes du même type sont alignés...