République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université des Sciences et de la Technologie Houari Boumediene

Faculté d'Informatique

Département d'Intelligence Artificielle et Sciences des Données

Rapport du projet vision par ordinateur :

Détection, Calibrage et Calcul de Position d'un Objet à l'aide de deux caméras Deux Caméras

MODULE : Vision par ordinateur

Réalisé par :

MIHOUBI Mohamed Anes
AITIDIR Abderahmane
TAOUCI Kenza
TAOURIRT Hamza

Année Universitaire : 2024/2025

Table des Matières

1 Introduction

Ce projet a pour objectif principal de concevoir et de développer un système capable de détecter, calibrer et calculer la position d'objets dans un espace tridimensionnel à l'aide de deux caméras. Le projet se décompose en plusieurs parties intégrant des approches théoriques et pratiques, tout en utilisant des outils avancés de traitement d'images.

L'architecture du système repose sur les étapes suivantes :

- **Détection d'objets** : Nous avons utilisé le modèle YOLO (You Only Look Once), une approche avancée de détection d'objets en temps réel, pour identifier des objets courants tels qu'un déodorant, et retourner leurs coordonnées dans le plan 2D.
- **Calibration des caméras** : Une méthode rigoureuse a été développée pour générer les matrices de paramètres intrinsèques et extrinsèques, essentielles pour corriger les distorsions optiques et établir une correspondance entre les points 3D et leurs projections 2D.
- **Calcul de la position et de la distance** : En combinant les résultats de la détection et de la calibration, nous avons calculé la position 3D des objets en exploitant des principes de stéréovision et de géométrie projective.
- **Améliorations supplémentaires** : Des fonctionnalités optionnelles ont été explorées pour enrichir le système, notamment une gestion dynamique de la calibration et DeepSORT pour le suivi multi-objets,.

Ce rapport détaille les aspects théoriques et techniques des différentes étapes du projet, en mettant en lumière les méthodes utilisées, les résultats obtenus, ainsi que les améliorations proposées. Les sections suivantes décrivent chaque étape en détail, avec des illustrations et des explications mathématiques adaptées.

2 Détection avec YOLOv9 sur un Dataset Personnalisé

2.1 Présentation de la Méthode

YOLOv9 (*You Only Look Once*, version 9) est un algorithme de détection d'objets en temps réel basé sur des réseaux de neurones convolutifs (CNN). Cet algorithme repose sur une seule passe pour localiser et classifier les objets dans une image, ce qui le rend très rapide et efficace par rapport aux méthodes plus traditionnelles.

L'entraînement sur un dataset personnalisé consiste à adapter le modèle pré-entraîné à un nouvel ensemble de données spécifiques, ici des déodorants, en fine-tuning. Ce processus permet au modèle de détecter efficacement les objets dans des scénarios qui ne sont pas inclus dans les datasets standards. (Pour plus de détails sur les étapes d'entraînement, un fichier `train_yolov9_object_detection_on_custom_dataset.ipyn` est fourni).

2.2 Étapes de Préparation et d'Entraînement

2.2.1 Préparation du Dataset

- **Acquisition des Images :** Un ensemble de 400 images réelles représentant des déodorants de différentes marques, tailles, couleurs, fonds et positions a été capturé.
- **Annotation Manuelle :** L'outil `LabelImg` a été utilisé pour annoter les images, générant des fichiers `.txt` pour chaque image, contenant les coordonnées normalisées des objets détectés.
- **Division des Données :** Le dataset a été divisé en trois sous-ensembles selon une répartition classique :
  - 80% pour l'entraînement.
  - 10% pour la validation.
  - 10% pour le test.

2.2.2 Entraînement Initial

Le modèle YOLOv9 a été entraîné avec les données préparées, en variant le nombre d'époques pour évaluer l'impact sur les performances.

2.2.3 Augmentation des Données

Pour améliorer la robustesse et la précision du modèle, diverses techniques d'augmentation des données ont été appliquées :

- **Flip :** Horizontal et Vertical.
- **Rotation :** $\pm 90°$ dans le sens horaire et antihoraire.
- **Shear :** $\pm 10°$ horizontalement et verticalement.
- **Grayscale :** Appliqué à 15% des images.
- **Brightness :** Variations de -22% à +22%.
- **Blur :** Jusqu'à 2.5px.
- **Noise :** Ajouté sur 0.93% des pixels.

2.2.4 Résultats

- **Précision sans augmentation :** 93%.
- **Précision avec augmentation :** 97%.

2.3 Configuration Nécessaire

2.3.1 Environnement de Développement

Google Colab a été utilisé pour l'entraînement en raison de la nécessité d'un GPU performant. Roboflow a été employé pour diviser le dataset et comparer les performances des modèles YOLO.

2.3.2 Bibliothèques et Outils

Les bibliothèques suivantes sont nécessaires pour exécuter le code :

- **Python :** Version 3.8 ou supérieure (problèmes d'incompatibilité avec 3.12).
- **PyTorch :** Framework pour l'entraînement des modèles de deep learning.
- **OpenCV :** Manipulation des images.
- **Matplotlib :** Visualisation des résultats.
- **Numpy :** Traitement des données.
- **LabelImg :** Annotation des données (phase de préparation).
- **Roboflow :** Gestion des données et augmentation.

Pour installer les bibliothèques nécessaires, utilisez :

```
pip install -r requirements.txt
```

(où `requirements.txt` est inclus dans le dossier YOLOv9).

2.4 Logique Théorique de YOLOv9

YOLO divise une image d'entrée en une grille et prédit simultanément :

- Les boîtes englobantes des objets (*bounding boxes*).
- Les probabilités de classification associées.

Chaque cellule de la grille est responsable de détecter un objet si son centre se trouve dans cette cellule. YOLOv9 améliore les versions précédentes grâce à :

- **Détection sans ancre (*Anchor-free detection*) :** Réduction des dépendances sur les ancres pré-définies.
- **Meilleures pyramides de caractéristiques :** Extraction de caractéristiques multi-échelles.
- **Module de mise au point (*Focus Module*) :** Amélioration de la vitesse et de la précision sur les objets complexes.

YOLO utilise une fonction de perte combinant les erreurs de localisation, de classification, et de dimension des boîtes pour optimiser les prédictions pendant l'entraînement.

3 Calibration

3.1 Principe Théorique

La calibration est basée sur le modèle de projection perspective, où une scène en 3D est projetée sur une image en 2D à l'aide des paramètres intrinsèques et extrinsèques d'une caméra. Ces paramètres comprennent :

3.1.1 Paramètres Intrinsèques

- La focale en pixels f_x, f_y.
- Le centre optique c_x, c_y.
- Les coefficients de distorsion radiale et tangentielle k_1, k_2, p_1, p_2.

3.1.2 Paramètres Extrinsèques

- La rotation R et la translation T entre le référentiel de la caméra et le référentiel du monde.

Le modèle de projection peut être exprimé par la formule suivante :

$$s \cdot \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K \cdot [R|T] \cdot \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Où :

- (X, Y, Z) sont les coordonnées 3D d'un point dans le monde réel.
- K est la matrice intrinsèque de la caméra.
- s est un facteur d'échelle.
- (u, v) sont les coordonnées projetées dans l'image 2D.

3.2 Fixation des Caméras

Deux caméras sont positionnées parallèlement avec un écart horizontal fixe appelé "baseline" (écart entre les centres de projection des caméras). Cet écart est désigné par b, et est mesuré en millimètres.

Pour une stéréovision correcte :

- Les caméras doivent être parfaitement alignées.
- L'orientation des caméras doit être identique pour minimiser les distorsions de parallaxe.

Dans le code, les caméras droite et gauche sont initialisées respectivement par `cv2.VideoCapture(1)` et `cv2.VideoCapture(2)`.

3.3 Étapes suivies :

1. **Capture des images** : Les images du damier sont capturées par les deux caméras.
2. **Détection des coins du damier** : La fonction `cv2.findChessboardCorners()` est utilisée pour détecter les coins dans l'image, qui servent de points de contrôle.
3. **Calibration dynamique** : La calibration est déclenchée automatiquement lorsque le damier est détecté dans le champ de vision d'une caméra. En cas de mouvement de la caméra, si le damier reste visible, un recalibrage est initié pour ajuster les paramètres. Cette approche garantit une calibration en temps réel.
4. **Calcul des paramètres** : Les points 2D extraits sont mis en correspondance avec les points 3D connus (les coordonnées du damier dans le monde réel). La fonction `cv2.calibrateCamera()` calcule les matrices intrinsèques et extrinsèques ainsi que les coefficients de distorsion.

3.4 Méthode développée

Pour obtenir les matrices de calibration :

1. Identifier les coins du damier en utilisant `cv2.findChessboardCorners()`.
2. Raffiner la position des coins avec `cv2.cornerSubPix()` pour une précision sous-pixellaire.
3. Stocker les points 3D du damier et leurs projections 2D détectées.
4. Calculer les matrices intrinsèques K, les coefficients de distorsion, et les paramètres extrinsèques R, T avec `cv2.calibrateCamera()`.

3.5 Erreur de Reprojection

L'erreur de reprojection mesure la précision de la calibration en comparant les points projetés avec les points observés. Elle est calculée comme suit :

$$\text{Erreur} = \frac{1}{N} \sum_{i=1}^{N} \|p_{\text{observé}}^{(i)} - p_{\text{projété}}^{(i)}\|^2$$

Où :

- N : Nombre total de points utilisés pour la calibration.
- $p_{\text{observé}}^{(i)}$: Coordonnées mesurées dans l'image.
- $p_{\text{projété}}^{(i)}$: Coordonnées projetées calculées à partir des points 3D et des paramètres calibrés.

Cette erreur est minimisée à l'aide de méthodes d'optimisation non linéaire, telles que l'algorithme de Levenberg-Marquardt, intégré dans `cv2.calibrateCamera()`.

3.6 Paramètres obtenus :

- **Matrice intrinsèque** :
$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$
f_x, f_y : distances focales ; c_x, c_y : coordonnées du centre optique.
- **Coefficients de distorsion** $[k_1, k_2, p_1, p_2]$: Corrigent les distorsions radiales et tangentielles.
- **Matrices extrinsèques** : R : Matrice de rotation ; T : Vecteur de translation, définissant la position et l'orientation de la caméra dans le monde.

4 Stéréovision

4.1 Détection des Points 2D

Une fois calibrées, les caméras capturent des images pour détecter la position 2D d'un objet ponctuel. L'objectif est de trouver les coordonnées (u_L, v_L) et (u_R, v_R) dans les images gauche et droite respectivement.

Pour garantir une correspondance correcte :

- Les coordonnées verticales v_L et v_R doivent être identiques.
- La différence horizontale $d = u_L - u_R$ (appelée disparité) est calculée.

4.2 Calcul de la Position 3D

La position 3D (X, Y, Z) de l'objet est calculée à l'aide de la stéréovision selon les équations suivantes :

$$Z = \frac{f_x \cdot b}{d}, \quad X = Z \cdot \frac{(u_L - c_x)}{f_x}, \quad Y = Z \cdot \frac{(v_L - c_y)}{f_y}$$

Où :

- f_x, f_y : distances focales.
- c_x, c_y : coordonnées du centre optique.
- b : baseline (distance entre les caméras).
- d : disparité horizontale entre les points correspondants des deux images.

4.3 Calcul du Point à Mi-Distance

Le point situé à mi-distance entre les centres de projection des deux caméras est calculé comme :

$$P_{\text{mid}} = \left(\frac{X_L + X_R}{2}, \frac{Y_L + Y_R}{2}, \frac{Z_L + Z_R}{2} \right)$$

Ce point fournit une référence pour des applications impliquant la synchronisation entre les caméras.

4.4 Implémentation en Temps Réel

Dans le code, la position 3D de l'objet est calculée en temps réel pour chaque trame capturée par les deux caméras. Les étapes incluent :

1. Capture simultanée des images par les deux caméras.
2. Envoi des trames à des sous-processus pour détection des objets et calibration.
3. Calcul des coordonnées 3D avec la méthode de stéréovision.
4. Affichage des résultats en temps réel.

5 Méthodologie

5.1 Détection et Suivi

Détection : YOLOv9 génère des boîtes englobantes (x_1, y_1, x_2, y_2) pour chaque objet.
Suivi : DeepSORT associe les détections entre frames, attribuant un ID unique à chaque objet.
Représentations :

- **YOLOv9 :** Coins supérieurs gauche (x_1, y_1) et inférieur droit (x_2, y_2).
- **DeepSORT :** Centre (c_x, c_y), largeur (w) et hauteur (h).

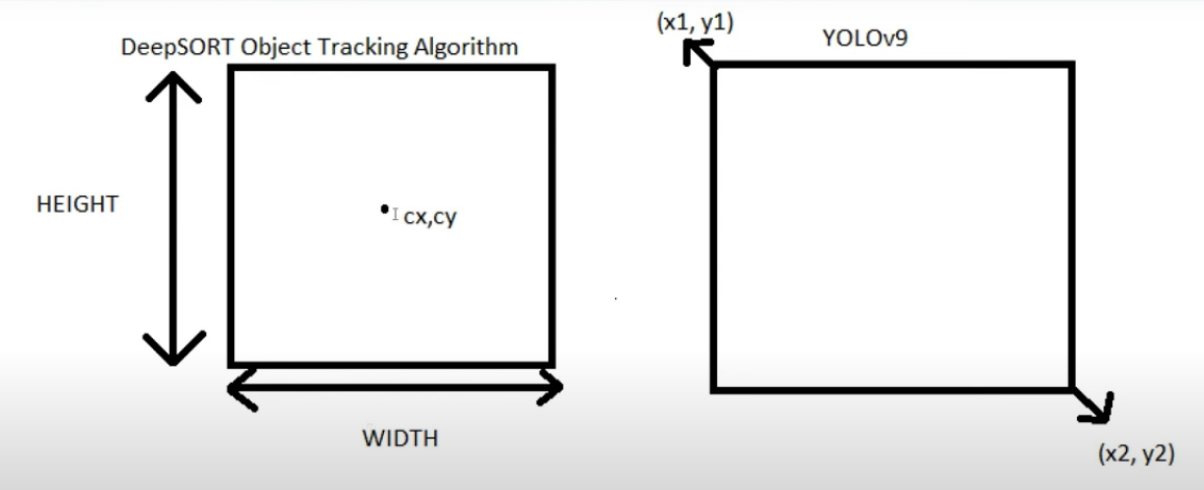

Figure 1: Représentation de YOLOv9 et DeepSORT.

5.2 Conversion entre Représentations

Les formules suivantes permettent de passer d'une représentation à l'autre :
De (x_1, y_1, x_2, y_2) **à** (c_x, c_y, w, h) **:**

$$c_x = \frac{x_1 + x_2}{2}, \quad c_y = \frac{y_1 + y_2}{2}, \quad w = x_2 - x_1, \quad h = y_2 - y_1$$

De (c_x, c_y, w, h) **à** (x_1, y_1, x_2, y_2) **:**

$$x_1 = c_x - \frac{w}{2}, \quad y_1 = c_y - \frac{h}{2}, \quad x_2 = c_x + \frac{w}{2}, \quad y_2 = c_y + \frac{h}{2}$$

5.3 Calcul de la Position et de la Vitesse

Position : Les positions successives sont stockées pour tracer les trajectoires.
Vitesse : Calculée à partir de la distance entre deux positions successives et de l'intervalle de temps :

$$\text{Distance} = \sqrt{(\Delta x)^2 + (\Delta y)^2}, \quad \text{Vitesse} = \frac{\text{Distance}}{\text{Intervalle de temps}}$$

5.4 Prédiction de Positions

Pour les objets hors champ, la prédiction est basée sur la vitesse :

$$\text{Position\_prédite} = \text{Position\_actuelle} + (\text{vitesse} \times \Delta t)$$

5.5 Mise en œuvre

Téléchargement :

- `easydict` et `gdown`.
- `deep_sort_pytorch` depuis Google Drive.

Décompression :

- Utilisation de `zipfile` pour décompresser `deep_sort_pytorch`.

Corrections pour les erreurs d'incompatibilité :

- Remplacer `np.int` par `np.int32` ou `np.int64` dans `deep_sort_pytorch/deep_sort/deep_sort.py`.
- Remplacer `np.float` par `np.float32` ou `np.float64` dans `deep_sort/utils.py`.

Les erreurs rencontrées proviennent de l'absence d'une version de DeepSORT-PyTorch spécifiquement adaptée à YOLOv9. Le développement de DeepSORT s'est arrêté à YOLOv5.

5.6 Résultats

- Trajectoires continues, même en cas de pertes temporaires.
- Suivi précis avec identification unique des objets.
- Estimation dynamique de la vitesse et de la distance.

5.7 Conclusion

Cette méthode offre un suivi robuste et précis des objets en mouvement, permettant une analyse détaillée de leurs trajectoires et de leurs vitesses. Les adaptations apportées à DeepSORT assurent une compatibilité avec YOLOv9 et une meilleure stabilité du système.

6 Conclusion

Ce projet a permis de concevoir et de mettre en œuvre un système avancé de détection, de suivi et de calcul de positions 3D d'objets en mouvement. En s'appuyant sur des techniques de pointe telles que **YOLOv9** pour la détection d'objets en temps réel et **DeepSORT** pour le suivi multi-objets, nous avons obtenu des résultats précis et robustes.

La **calibration des caméras** a été rigoureusement réalisée pour assurer la cohérence des projections 2D et des calculs de positions 3D grâce à la stéréovision. Cette étape, combinée à des méthodes d'optimisation comme l'erreur de reprojection, a renforcé la fiabilité du système.

Les résultats obtenus montrent :

- Une détection efficace des objets personnalisés (précision améliorée grâce à l'augmentation des données).
- Un suivi continu et précis même en cas de pertes temporaires.
- Une estimation dynamique des trajectoires et vitesses des objets.

Malgré les défis rencontrés, notamment l'adaptation de **DeepSORT-PyTorch** pour fonctionner avec **YOLOv9**, les corrections apportées ont assuré une compatibilité optimale et une stabilité améliorée.

En conclusion, ce système démontre un potentiel significatif pour des applications en temps réel nécessitant une détection et un suivi précis, comme la robotique, la surveillance intelligente ou la réalité augmentée. Des travaux futurs pourraient explorer l'intégration d'algorithmes plus sophistiqués pour la prédiction des trajectoires et l'optimisation du traitement en temps réel.

Bibliographie

- **DeepSORT:** disponible à https://drive.google.com/uc?id=11ZSZcG-bcbueXZC3rN08CM0qqX3eiHxfconfirm=t, 2024. Accédé en décembre 2024.
- **YOLOv9:** disponible à https://github.com/SkalskiP/yolov9.git, 2024. Accédé en décembre 2024.
- **LabelImg:** disponible à https://github.com/tzutalin/labelImg, 2024. Accédé en décembre 2024.