Détection des contours de pièces de puzzle – Résumé d'analyse

Chargement des bibliothèques et de l'image

Le notebook commence par importer les bibliothèques essentielles au traitement d'image : OpenCV (`cv2`), NumPy (`numpy`) pour les calculs sur matrices, et Matplotlib (`matplotlib.pyplot`) pour l'affichage. L'image contenant les pièces est chargée à l'aide de `cv2.imread`. Pour un affichage correct avec Matplotlib, l'image est convertie de BGR vers RGB avec `cv2.cvtColor`. Une conversion supplémentaire en HSV est réalisée pour faciliter les opérations basées sur la couleur (notamment la teinte et la saturation).

Définition d'une région de référence (ROI) et histogramme

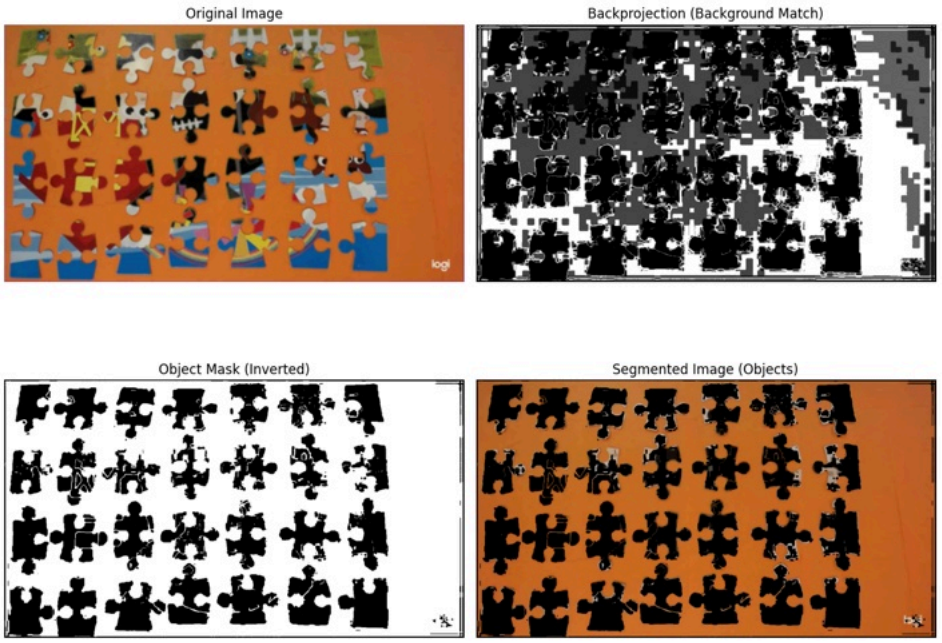
Une **région d'intérêt** (ROI) dans le fond de l'image (souvent orange) est sélectionnée manuellement. À partir de cette zone, un **histogramme 2D** des composantes Hue et Saturation est calculé avec `cv2.calcHist`, puis normalisé à l'aide de `cv2.normalize`. Cet histogramme représente la distribution des couleurs du fond et servira pour identifier les zones similaires dans toute l'image.

Rétroprojection et génération d'un masque binaire

L'histogramme est projeté sur l'image entière en espace HSV via `cv2.calcBackProject`. Cette opération produit une image en niveaux de gris où les pixels ressemblant au fond sont plus clairs. Pour réduire le bruit, un floutage est appliqué avec `cv2.filter2D` ou `cv2.GaussianBlur`. Ensuite, un **seuillage binaire** (`cv2.threshold`) permet de générer un masque : les zones correspondant au fond deviennent blanches, et les autres (potentiellement les pièces) noires. L'inversion du seuillage (`cv2.THRESH_BINARY_INV`) est parfois testée selon les cas.

Nettoyage du masque par morphologie

Le masque binaire initial peut comporter des artefacts ou des trous. Pour l'améliorer, des **opérations morphologiques**sont utilisées avec un noyau elliptique (`cv2.getStructuringElement`). Une **ouverture** (`cv2.morphologyEx` avec `MORPH_OPEN`) supprime les petits éléments indésirables. Une **fermeture** (`MORPH_CLOSE`) comble les trous éventuels à l'intérieur des zones blanches du masque. Le résultat final est un masque plus propre où le fond est clairement séparé des objets.

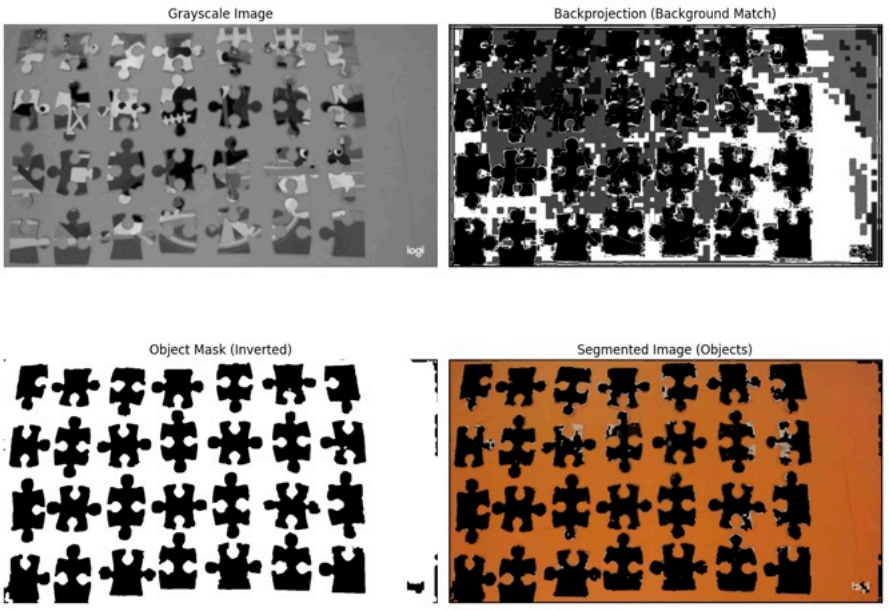


Application du masque et visualisation de la segmentation du fond

Une fois ce masque du fond obtenu, on l'applique sur l'image originale pour visualiser le résultat de la segmentation. La fonction `cv2.bitwise_and` est utilisée avec le masque : on combine l'image couleur d'origine avec elle-même en utilisant le masque binaire, de sorte que seules les zones où le masque est blanc sont conservées. Autrement dit, on **masque** les pièces (zones noires du masque) et on ne garde que le fond. Le résultat est une image où le fond orange apparaît normalement et tout le reste de l'image est noir. Cette étape permet de vérifier que le masque a correctement isolé le fond. La cellule affiche ensuite plusieurs images côte à côte à l'aide de Matplotlib: l'image originale, l'image de **backprojection** en niveaux de gris, le **masque binaire** obtenu, et l'**image segmentée** résultante. Chaque sous-figure est titrée (par exemple *Image originale*, *Backprojection*, *Masque binaire*, *Fond orange segmenté*) et les axes sont masqués (`plt.axis('off')`) pour ne garder que l'image. On constate sur ces affichages si les pièces de puzzle sont bien exclues du masque du fond.

Affinage du masque à l'aide de l'intensité en niveaux de gris (approche complémentaire)

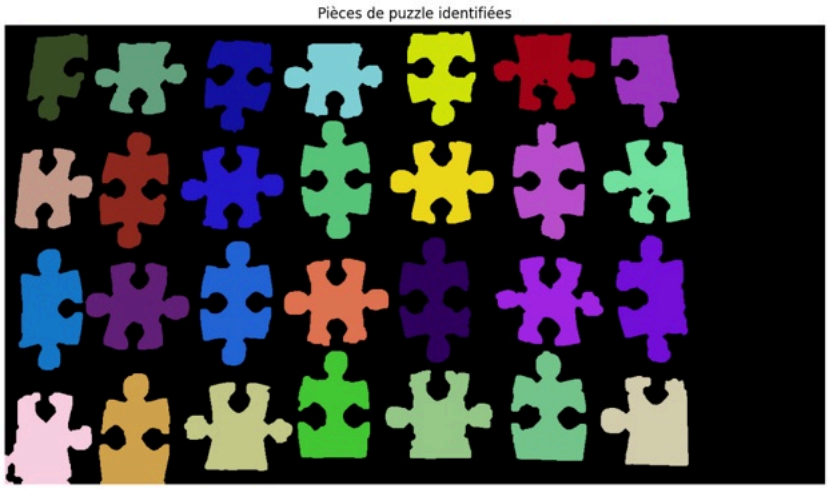
À ce stade, l'utilisateur a noté que le masque binaire pourrait être encore amélioré en tenant compte de la **luminance** de l'image. Une cellule du notebook convertit l'image en **niveau de gris** (intensité) ou extrait la composante Value (V, brillance) de l'image HSV, afin d'analyser la distribution des intensités des pixels. L'idée est de créer un **deuxième masque** basé sur l'image en niveaux de gris pour affiner la séparation pièces/fond. Par exemple, un histogramme des valeurs de brillance a pu être tracé pour déterminer des seuils d'intensité "trop sombres" ou "trop claires" qui ne correspondent pas aux pièces. L'utilisateur définit alors des seuils (par ex. `threshold_min = 100` et `threshold_max = 140`) et considère que les pixels du masque initial qui sont en dehors de cette plage de luminosité ne devraient pas être des pièces. Concrètement, le code met à zéro (noir) les pixels du masque initial dont la valeur en niveaux de gris est hors de l'intervalle [100,140] (ceci cible par exemple des reflets très clairs ou des ombres très foncées du fond qui auraient pu être marqués à tort comme pièces). Après cette suppression conditionnelle, de nouvelles opérations morphologiques sont appliquées – typiquement une **érosion puis une dilatation** successives (**`cv2.erode`** puis **`cv2.dilate`** avec un élément 3×3 répété quelques fois) – pour éliminer les petits artéfacts restants et **boucher les trous** à l'intérieur des zones de pièce. Le résultat est un **masque binaire affiné des pièces**, que l'on nomme `refined_mask`. Dans ce masque, on cherche à avoir les pièces de puzzle en blanc sur fond noir, propres et sans trous ni bruit. Une figure est affichée pour vérifier ce masque affiné, souvent annotée comme *Mask objet inversé* (pièces en blanc).



Inversion du masque et détection des composantes connectées (pièces)

Une fois le masque final des pièces obtenu, on s'assure que les pièces sont bien représentées en **blanc sur fond noir** (si nécessaire, on utilise **`cv2.bitwise_not(refined_mask)`** pour inverser le masque). Les pièces étant maintenant des objets blancs distincts, la cellule suivante identifie chaque pièce en tant qu'**objet distinct** à l'aide de la fonction **`cv2.connectedComponentsWithStats`**. Cette

fonction analyse l'image binaire et attribue un label unique à chaque ensemble de pixels blancs connectés (en 8-connexité). Elle fournit également des **statistiques** pour chaque composante, notamment l'aire en pixels et la **boîte englobante** (coordonnées X, Y du coin supérieur gauche et largeur, hauteur) de chaque composante. La cellule parcourt ces résultats : on récupère le nombre total de composants détectés (`num_labels`) et leurs attributs. Un filtrage est appliqué pour éliminer les composantes trop petites qui pourraient correspondre à du bruit restant : par exemple, un seuil minimum d'aire (**`min_area = 2000`** pixels) est fixé, et seules les composantes dont l'aire excède ce seuil sont considérées comme de vraies pièces de puzzle.



Visualisation des pièces identifiées sur l'image

Pour s'assurer de la bonne identification des pièces, le notebook génère une image de visualisation où chaque pièce détectée est affichée avec une couleur distincte. On crée d'abord une image vide (noire) de la taille de l'image originale en 3 canaux. Puis, pour chaque label de pièce retenu, on attribue une couleur aléatoire générée avec NumPy (**`np.random.randint(0, 255, size=(num_labels, 3))`**) et on colorie dans cette image de sortie tous les pixels correspondant à ce label. Chaque pièce apparaît ainsi d'une couleur différente, facilitant leur distinction visuelle. Cette image colorée des pièces segmentées est affichée avec Matplotlib (taille 10x6) et titrée *Pièces de puzzle identifiées*. On voit clairement chaque pièce de puzzle colorisée de manière unique sur fond noir.

En parallèle, une autre visualisation ajoute des **boîtes de délimitation (bounding boxes)** autour de chaque pièce directement sur l'image originale. Pour cela, la bibliothèque Matplotlib est utilisée avec son module de patches : on trace un rectangle (**`patches.Rectangle`**) rouge pour chaque boîte englobante (aux coordonnées X, Y et de dimensions w×h fournies par `connectedComponentsWithStats`). Un texte d'identification (par exemple "ID 3") est également placé près de chaque rectangle en jaune pour numéroter les pièces. Cette figure donne un aperçu de l'emplacement et de l'étendue de chaque pièce sur l'image de départ. Elle est affichée avec le titre *Bounding Boxes of Puzzle Pieces*, sans axes.

Extraction et sauvegarde individuelle de chaque pièce

Le notebook procède ensuite à l'extraction de chaque pièce isolée. Une nouvelle structure de données (dictionnaire `puzzle_pieces`) est remplie en parcourant les labels de pièces détectées. Pour chaque pièce valide :

- On récupère sa boîte englobante `(x, y, w, h)` à partir des statistiques calculées.
- On extrait le **masque binaire de la pièce** en isolant les pixels de l'étiquette correspondante (par exemple en créant une image `mask_piece` où les pixels ayant le label *i* sont mis à 255). Ce masque de la pièce est ensuite **rogné** aux dimensions de la boîte englobante pour ne garder que la zone autour de la pièce (`piece_mask = mask_piece[y:y+h, x:x+w]`).
- On extrait de l'image originale la sous-image correspondant à la pièce : `piece_img = original_img[y:y+h, x:x+w]`. Il s'agit de la zone couleur de la pièce, incluant encore le fond autour mais recadrée.
- On applique de nouveau un masquage bit-à-bit pour ne conserver que la pièce sur son fond local : **`cv2.bitwise_and(piece_img, piece_img, mask=piece_mask)`**. Ceci met à noir tous les pixels autour de la pièce, ne laissant que la pièce de puzzle en couleur sur fond noir dans l'image résultante (`piece_masked`).

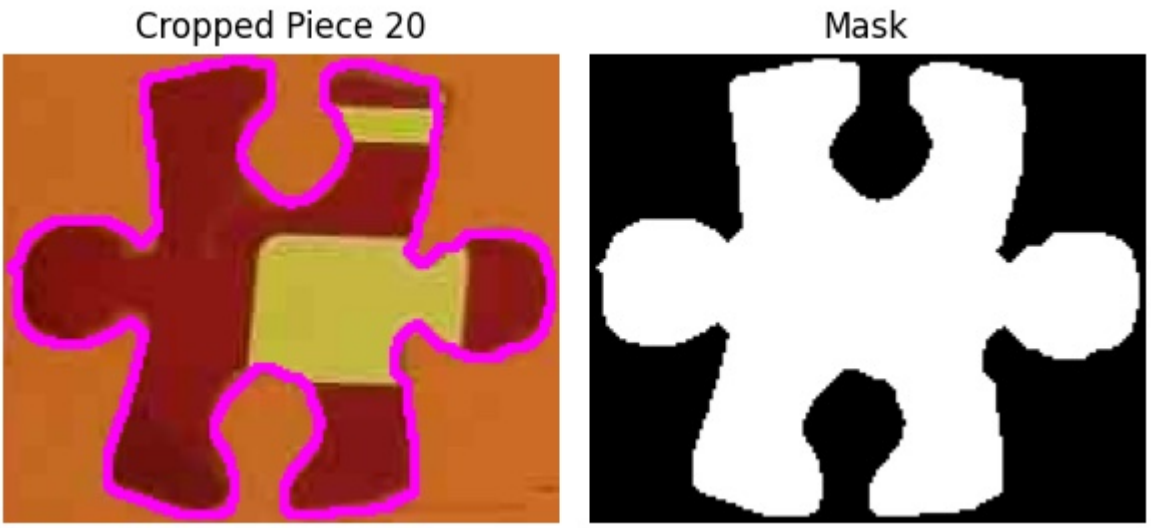
Ces informations sont sauvegardées : le dictionnaire `puzzle_pieces` associe à un identifiant de pièce un ensemble de données (la boîte englobante, le masque binaire de la pièce, l'image couleur de la pièce isolée, et l'aire en pixels). À la fin de ce processus, on peut consulter la liste des pièces extraites (par exemple en affichant `puzzle_pieces.keys()`, on voyait les identifiants 0, 1, 2, ..., 27 dans le cas présent, ce qui indique 28 pièces détectées et stockées).

Recherche des contours de chaque pièce (détection des formes)

La dernière étape tentée dans le notebook est la **détection de contours** sur chaque pièce de puzzle isolée, dans le but d'analyser leur forme. Pour ce faire, le code utilise **`cv2.findContours`** sur le masque binaire de la pièce. Chaque masque de pièce (`piece_mask`) est une petite image en niveaux de gris (255 pour la pièce, 0 pour le fond). La fonction `cv2.findContours(piece_mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)` est appelée : on spécifie le mode `RETR_EXTERNAL` pour ne détecter que le contour externe (les pièces n'ayant pas de trous internes à détecter, a priori) et la méthode `CHAIN_APPROX_SIMPLE` pour obtenir une approximation des contours (cette méthode économise de la mémoire en simplifiant la suite de points du contour). La fonction retourne la liste des contours trouvés (ici normalement un seul contour principal par pièce).

Ensuite, **`cv2.drawContours`** est utilisé pour tracer ce contour sur l'image de la pièce. On dessine sur l'image couleur de la pièce isolée (`piece_masked`) tous les contours détectés,

avec une couleur visible (par exemple *magenta* `(255, 0, 255)`) et une épaisseur de trait de 2 pixels. Pour visualiser le résultat pour chaque pièce, le notebook affiche côte à côte l'image de la pièce avec son contour superposé et le masque binaire correspondant. Une figure Matplotlib est créée pour chaque pièce : à gauche la pièce en couleur avec le contour dessiné, à droite son masque en niveaux de gris, ce qui permet de comparer la forme. Chaque figure est titrée (par ex. *Cropped Piece 20* et *Mask* pour la pièce d'ID 20) et on masque les axes pour une lecture claire.



Conclusion et limites de l'analyse

En suivant les étapes ci-dessus, le notebook parvient à **segmenter toutes les pièces de puzzle** depuis l'image initiale, à les isoler sur fond noir et à tracer le contour extérieur de chacune. Cependant, l'utilisateur souligne une **limite** de son approche : il n'a pas réussi à exploiter ces contours de manière suffisamment poussée pour **distinguer les pièces par leur forme**. En effet, si les contours des pièces sont bien extraits et dessinés, aucune analyse plus avancée (par exemple comparaison de formes, identification de motifs d'encoches des puzzles, etc.) n'a été réalisée. Les contours obtenus restent basiques et ne permettent pas en l'état de différencier ou classifier les pièces uniquement via leur silhouette. Il faudrait des traitements supplémentaires (analyse de forme, description par des descripteurs, etc.) pour reconnaître chaque pièce par sa forme particulière, ce qui n'a pas pu être accompli dans ce notebook. Ainsi, le résultat final se limite à une segmentation et extraction des pièces, sans aller jusqu'à la reconnaissance formelle de leur profil.