Rapport PFE - Reconstitution automatique de puzzles

Andrea Millasseau - Jason Morisse - Matthieu Nass - Thomas Thorignac

27 mars 2024

université de BORDEAUX

Table des matières

Introduction (contexte, problématique, sujet)

Contexte

100 000 tonnes de jeux de société sont jetés chaque année et leur durée d'utilisation est à peine de 8 mois. C'est pour cela que Deuzio a été créée. Il s'agit d'une entreprise spécialisée dans le contrôle, le reconditionnement et la revente de jeux de seconde main (jeux de société, legos, puzzles...).

Problématique

Une des étapes clé lors du contrôle d'un puzzle est de vérifier sa complétude, ce qui est pour l'instant fait manuellement. Cette étape pourrait être simple si chaque puzzle avait exactement le nombre de pièces annoncé sur la boîte, mais ce n'est pas le cas (e.g. une boîte de 200 pièces peut avoir par exemple 196 ou 204 pièces). Cette étape est donc compliquée, mais aussi très chronophage en fonction du nombre de pièces. C'est pourquoi cette vérification n'est faite pour l'instant que sur des puzzles de 100 pièces maximum au sein de l'entreprise.

Sujet

Le projet consiste donc à réaliser un outil de traitement, de synthèse et d'analyse d'images permettant de reconstruire automatiquement un puzzle, une fois ses pièces toutes dans le même sens, afin de vérifier s'il est complet et si le jeu peut donc être proposé à nouveau à la vente.

Après discussion avec le client, nous avons conclu que le logiciel doit être simple d'utilisation, rapide à exécuter, utilisable sur du matériel relativement ancien fonctionnant sous iOS et doit être assez fiable pour être utilisé dans un contexte d'entreprise.

1 État de l'art / Existant

1.1 Point de départ/d'inspiration

Il nous a été fourni dans notre sujet de PFE, un projet "point de départ ou d'inspiration" : PuzzleSolver 2019, Ken Ellinwood

Ce programme reprend le travail original de Joe Zeimen de 2013. C'est une application hybride ligne de commande/interface utilisateur. Elle possède deux modes de fonctionnement :

— *mode automatique :* résout le puzzle automatiquement, fonctionnel pour des puzzles complets d'environ 100 pièces ou moins.
— *mode guidé :* mode interactif, permet de résoudre des puzzles avec un grand nombre de pièces, ou des puzzles partiellement complétés.

Les lignes de commandes sont utilisées pour prendre en compte les paramètres d'entrées, l'interface utilisateur apparaît pour ajuster les coins d'une pièce de puzzle, vérifier les contours d'une pièce, ou, avec le mode d'utilisation guidé, de valider ou non la correspondance d'une paire de pièces.

Processus :

— numéroter les pièces (en partant de 1)
— utiliser un scanner à plat pour scanner le dos des pièces (placées dans l'ordre de gauche à droite, de haut en bas)
— mettre toutes les images dans un répertoire
— lancer l'application avec le répertoire en paramètre (en changeant les paramètres jusqu'à ce que toutes les pièces soit reconnues et les *warning* concernant la qualité des coins disparaissent)
— choisir le mode de solution

Fonctionnement :

— Les images sont filtrées pour réduire le bruit (par défaut, la fonction `medianBlur()` de `OpenCV`.
— La fonction `findContours()` de `OpenCV` est ensuite utilisée pour détecter les contours des pièces (passées préalablement en noir et blanc).
— Les coins des pièces sont localisés en utilisant un processus itératif qui appelle la fonction `goodFeaturesToTrack()` de `OpenCV`. Un contrôle de qualité est ensuite effectué sur les coins.
— Le contour de chaque pièce est ensuite divisé en 4 bords, qui sont alors analysés et classifiés en trois types : `OUTER_EDGE`, `TAB` ou `HOLE`.
— Le solveur calcule ensuite des scores pour chaque combinaison bord-bord possibles, plus le score est proche de 0, plus il y a de chance que les pièces soient compatibles.

Ce code n'est pas libre de droit, nous avons donc décidé de ne pas l'utiliser, mais nous nous sommes inspirés des algorithmes utilisés pour la similarité

des bords entre-eux.

1.2 Autres ressources

1.2.1 Jigsaw Puzzle solver using C++, SIFT and FLANN in `OpenCV`

— *SIFT (Scale-Invariant Feature Transform) :* création de *features* distinctes avec des *keypoints* qui ne changent pas en fonction de l'échelle (par exemple, un coin dans une petite image peut s'aplatir et ne plus être détecté si un zoom est effectué)

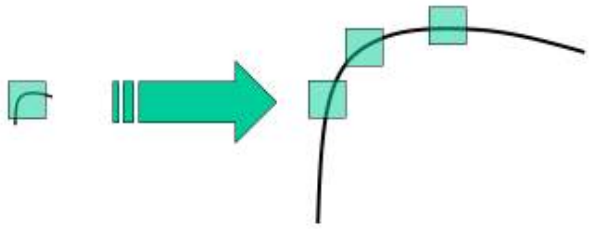


FIGURE 1 – Détection d'angle avec zoom

L'algorithme est expliqué en détail dans cet article [2]

— *FLANN (Fast Library for Approximate Nearest Neighbors) :* collection d'algorithmes optimisés pour une recherche rapide du voisin le plus proche dans un grand *dataset* avec une grande dimension de *features*

Un Solver de 2012 fait par David Lavy : Seuillage d'image pour isoler les pièces du fond, détection de contours stockés dans un tableau, extraction de chaque pièce en images individuelles, utilisation de *SIFT* pour chaque pièce, correspondance des *keypoints* de SIFT en utilisant FLANN, filtrage en fonction de la distance euclidienne. Encore à faire : positionner les contours des pièces sur l'image du puzzle.

Note : Nous n'avons pas réussi à compiler ce programme et la vidéo tutoriel de l'implémentation n'est pas disponible, nous n'avons donc pas pu tester ce code.

1.2.2 Solving Jigsaw puzzles with `Python` and `OpenCV` [1]

Le code source peut être trouver ici. Description du programme :

— création d'un *dataset* (200+ photos de pièce individuelle dans des conditions d'éclairage optimal)

— segmentation

— détecter les 4 coins de pièces
— séparation des 4 bords
— (implémentation non finie)

1.2.3 Aguiar, M. (2022). Jigsaw Puzzle Solver (Doctoral dissertation, WORCESTER POLYTECHNIC INSTITUTE)

Cet article [3] détaille plusieurs méthodes (contours, histogramme de couleur, SIFT, ...) et fait un design hybride d'aide à l'utilisateur pour résoudre des puzzles. Il montre quelles caractéristiques sont les plus importantes en fonction du puzzle (e.g. meilleurs résultats avec les bords pour un puzzle monochrome, meilleurs résultats avec les couleurs pour un puzzle en dégradé de couleurs).

Il nous a donné des idées d'algorithmes de correspondance entre les pièces en utilisant différentes caractéristiques :
— les bords
— les traits internes aux pièces
— la couleurs des pièces

1.3 Vidéos

Nous avons également exploré quelques vidéos pour nous inspirer et nous donner des idées de pistes que l'on pourrait explorer.
— Worlds hardest jigsaw vs. puzzle machine (all white), Stuff Made Here
— Solving Jigsaw Puzzle Using Computer Vision..., Rujuta Vaze

2 User Stories

— Prendre des photos directement depuis l'application/importation de photos depuis la galerie vers l'application, pour les utiliser ensuite pour des traitements
— Vérification manuelle par l'utilisateur de la segmentation effectuée sur les photos des pièces de puzzle
— Résolution (ou confirmation de son impossibilité) par l'application, avec vérification (visualisation du puzzle reconstruit) par l'utilisateur

3 Besoins fonctionnels, non fonctionnels, tests

3.1 Réalisation d'une application IOS

Le projet final sera rendu sous la forme d'une application IOS. Initialement, le client demandait une application pour son iPad, mais le modèle est encore sous IOS 12, et nous n'avons pas pu trouver un moyen de réaliser une application avec cette version du système d'exploitation. Il a donc été convenu avec le client que nous allions réaliser l'application pour une version d'IOS plus récente, qu'il pourrait utiliser sur son téléphone.

3.2 Prise / Importation de photos

L'application doit permettre soit de prendre des photos directement soit de pouvoir importer des photos (ou les deux). Il est spécifié dans le sujet du projet que le matériel utilisé serait préférablement un iPad, le format d'image doit donc être adapté à cet appareil.

3.3 Précision des résultats

Étant donné que le but du projet est de pouvoir vérifier la complétude d'un puzzle (avec comme intention de pouvoir le revendre), il est impératif d'avoir une précision de 100% dans le cas ou un puzzle est détecté comme étant complet, le cas contraire mettant potentiellement un objet défectueux à la vente.

3.4 Vérification manuelle des résultats (visualisation)

Nous avons sus-mentionné qu'il est impératif d'avoir une très haute précision. Il est quasiment impossible d'avoir une précision de 100%, car il ne suffit pas de compter le nombre de pièces. Par conséquent, une vérification manuelle des résultats obtenus est donc nécessaire pour confirmer ces résultats. Un exemple serait l'affichage du puzzle terminé une fois reconstruit par le programme, permettant une vérification de l'utilisateur.

3.5 Détection des pièces

Une fois la photo des pièces de puzzles prise/importées dans l'application, il faut effectuer une segmentation sur la photo pour permettre de distinguer les pièces entre elles. Il faut également prendre en compte qu'il est possible

que deux pièces se "chevauchent" et pouvoir détecter ce cas particulier, qui pourrait poser des problèmes si cela n'était pas le cas.

3.6 Rogner l'image d'entrée

L'image passée en entrée est traitée via une segmentation réalisée par un CNN (réseau de neuronal convolutif) pour détecter les pièces de puzzles et leurs contours. Il est possible que la photo ne soit pas bien cadrée, et les bords de la table sur laquelle sont posées les pièces pourraient poser problème (ils pourraient être détectés comme étant des contours de pièces). Une fonctionnalité permettant de rogner l'image pour ne garder que la partie de la table avec les pièces, sans les bords, est donc une bonne option.

4 Choix logiciels ARGUMENTÉS (bibliothèques, langage, ...)

4.1 Bibliothèque `OpenCV`

La bibliothèque `OpenCV` est couramment utilisée pour de la segmentation et du traitement d'images, puisqu'elle comprend des algorithmes essentiels pour cela, comme des algorithmes de détection de contours, filtres de couleur, etc...

Elle est également utilisée dans beaucoup de ressources que nous avons trouvées sur le sujet (ainsi que dans la base de code fournie dans le sujet).

4.2 Segmentation d'image avec un modèle de type Unet

Pour notre projet, il nous a semblé impératif d'utiliser de la segmentation d'image, le plus précise possible. Nous avons choisi d'utiliser un modèle de type UNet, un réseau neuronal convolutif (ou CNN), pour nous permettre une détection par segmentation précise, sur des photos comportant un nombre élevé de pièces (la segmentation d'image basique étant moins performante sur de petits objets).

Pourquoi avoir choisi le modèle Unet[4] ?
Les modèles de type UNet sont adaptés à de la segmentation d'images et pour la détection et la localisation précise d'objets dans une scène. L'architecture (connexions directes entre l'encodeur et le décodeur) aide à minimiser la dégradation du signal pour faciliter l'apprentissage. Ce modèle est relativement léger en terme de nombre de paramètres, par rapport à d'autres modèles de *deep learning*, permettant d'avoir un faible coût computationnel.

4.3 Interface Graphique : `Tkinter`

Tous le code que nous avons rédigé ayant été fait en python, il était donc logique pour nous de choisir une bibliothèque python pour réaliser une interface graphique.

Nous avons choisi d'utiliser `Tkinter` car il s'agit d'une bibliothèque pratique et facile à appréhender et utiliser pour de l'affichage d'image et un peu d'interactions avec.

5 Architecture

5.1 Pipeline

La figure suivante que nous avons réalisé sous la forme d'un *pipeline*, correspond à l'ensemble des étapes que nous avons mises en place, en partant de l'acquisition des pièces de puzzle, jusqu'à la reconstitution du puzzle.

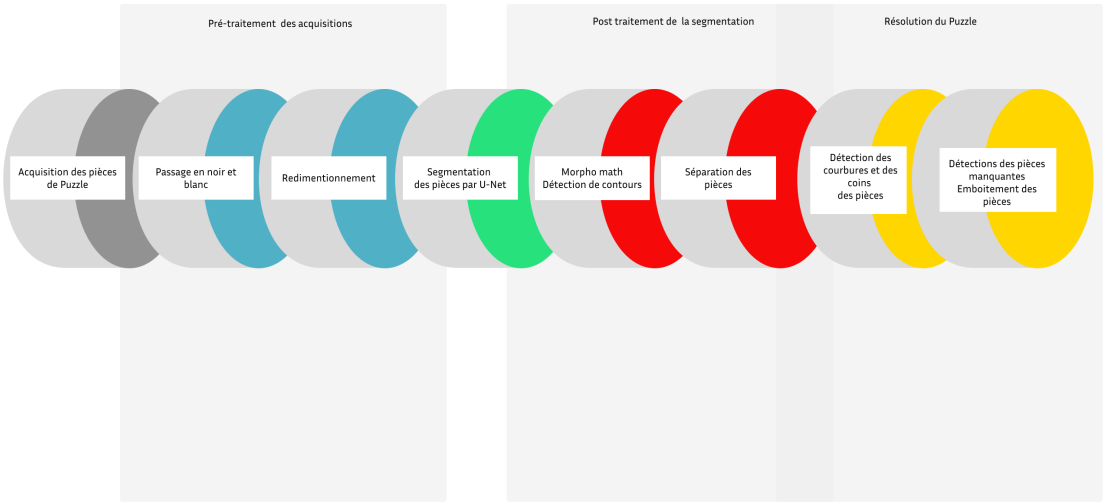


FIGURE 2 – Pipeline générale de l'application

5.2 Arborescence

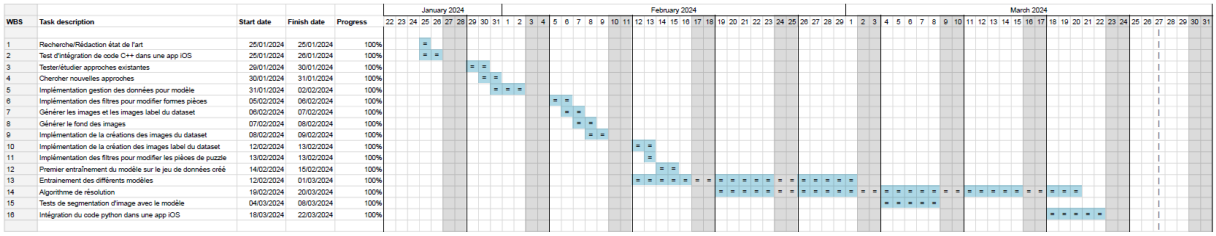
Voici ci-dessous l'arborescence des fichiers qui servent à alimenter l'application. Le fichier `App.py` est le fichier principal, et le reste du code est principalement contenu dans le répertoire `modules`.

```
App.py

modules
|       AnimatedGIF.py
|       augment_light.py
|       auxiliars.py
|       crop.py
|       debruitage.py
|       drawmath.py
|       findcorners.py
|       maths.py
|       missing_packages.py
|       resolution_algorithme.py
|       rgb_to_hsv.py
|       separatePieces.py

model
|       __init__.py
|       unet_model.py
|       unet_parts.py
|       MultiDim_BW_60_epochs # Modèle UNet pour segmentation

images
|       flash_labels
        |       # Contient les images des labels des pièces de puzzle

GIF_image
|       loading.gif
```

FIGURE 3 – Arborescence des fichiers de l'application

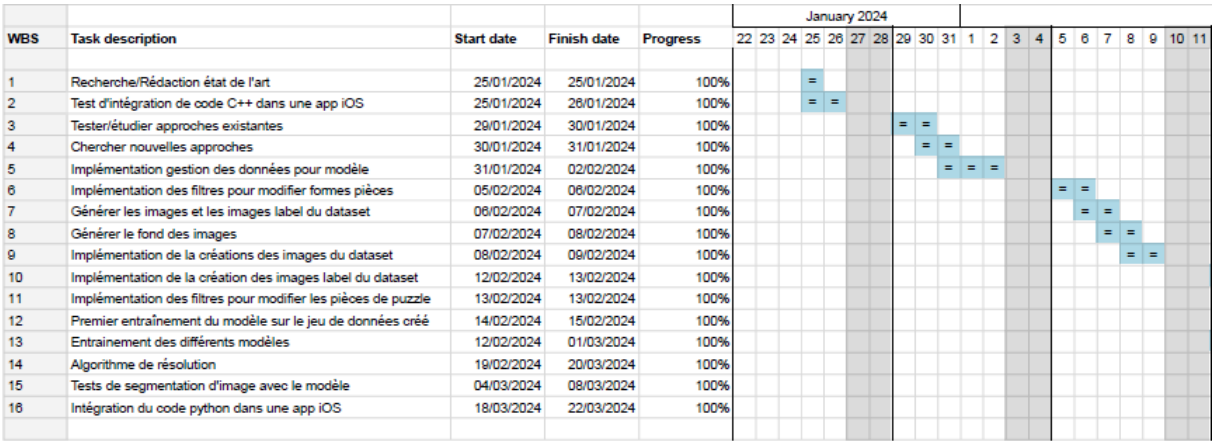
Cependant, tout le travail effectué n'étant pas intégré dans l'application pour différentes raisons (manque de temps, problème d'exploitation de résultats pour effectuer les traitements suivants, ...), notre dépôt Git contient d'autres répertoires contenant du code sous forme de fichiers notebook, avec des exemples d'utilisation du code en vue de potentiels travaux futurs.

6 Gantt prévisionnel



WBS	Task description	Start date	Finish date	Progress
1	Recherche/Rédaction état de l'art	25/01/2024	25/01/2024	100%
2	Test d'intégration de code C++ dans une app iOS	25/01/2024	26/01/2024	100%
3	Tester/étudier approches existantes	29/01/2024	30/01/2024	100%
4	Chercher nouvelles approches	30/01/2024	31/01/2024	100%
5	Implémentation gestion des données pour modèle	31/01/2024	02/02/2024	100%
6	Implémentation des filtres pour modifier formes pièces	05/02/2024	06/02/2024	100%
7	Générer les images et les images label du dataset	06/02/2024	07/02/2024	100%
8	Générer le fond des images	07/02/2024	08/02/2024	100%
9	Implémentation de la créations des images du dataset	08/02/2024	09/02/2024	100%
10	Implémentation de la création des images label du dataset	12/02/2024	13/02/2024	100%
11	Implémentation des filtres pour modifier les pièces de puzzle	13/02/2024	13/02/2024	100%
12	Premier entraînement du modèle sur le jeu de données créé	14/02/2024	15/02/2024	100%
13	Entrainement des différents modèles	12/02/2024	01/03/2024	100%
14	Algorithme de résolution	19/02/2024	20/03/2024	100%
15	Tests de segmentation d'image avec le modèle	04/03/2024	08/03/2024	100%
16	Intégration du code python dans une app iOS	18/03/2024	22/03/2024	100%

FIGURE 4 – Gantt prévisionnel



WBS	Task description	Start date	Finish date	Progress
1	Recherche/Rédaction état de l'art	25/01/2024	25/01/2024	100%
2	Test d'intégration de code C++ dans une app iOS	25/01/2024	26/01/2024	100%
3	Tester/étudier approches existantes	29/01/2024	30/01/2024	100%
4	Chercher nouvelles approches	30/01/2024	31/01/2024	100%
5	Implémentation gestion des données pour modèle	31/01/2024	02/02/2024	100%
6	Implémentation des filtres pour modifier formes pièces	05/02/2024	06/02/2024	100%
7	Générer les images et les images label du dataset	06/02/2024	07/02/2024	100%
8	Générer le fond des images	07/02/2024	08/02/2024	100%
9	Implémentation de la créations des images du dataset	08/02/2024	09/02/2024	100%
10	Implémentation de la création des images label du dataset	12/02/2024	13/02/2024	100%
11	Implémentation des filtres pour modifier les pièces de puzzle	13/02/2024	13/02/2024	100%
12	Premier entraînement du modèle sur le jeu de données créé	14/02/2024	15/02/2024	100%
13	Entrainement des différents modèles	12/02/2024	01/03/2024	100%
14	Algorithme de résolution	19/02/2024	20/03/2024	100%
15	Tests de segmentation d'image avec le modèle	04/03/2024	08/03/2024	100%
16	Intégration du code python dans une app iOS	18/03/2024	22/03/2024	100%

FIGURE 5 – Gantt prévisionnel partie gauche

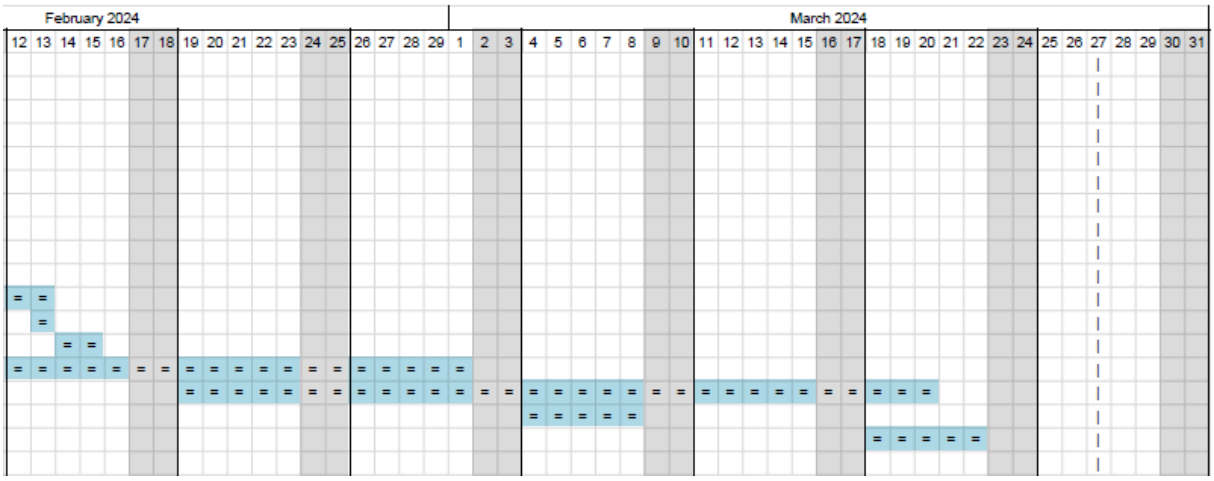


FIGURE 6 – Gantt prévisionnel partie droite

Le Gantt a été séparé en 2 parties pour une meilleure lisibilité.

7 Réalisation

7.1 Interface graphique

Il était initialement question dans notre projet de réaliser une application pour iOS. Au fil du projet, nous nous sommes rendu compte qu'il serait difficile de réaliser une telle application, notamment par le fait que nous n'avions pas trouvé de solution pour ajouter du code python dans une application `Swift`, mais aussi par manque de temps.

Nous avons donc pris la décision de réaliser une interface graphique simple avec la bibliothèque python `Tkinter`. Voici la fenêtre principale de notre interface graphique.

FIGURE 7 – Fenêtre principale de l'interface graphique

Sur cette fenêtre, différents boutons apparaissent. Nous allons entrer plus en détail dans les fonctionnalités dans les sections suivantes.

7.1.1 Importer des images

Le premier bouton permet, à partir des répertoires de l'ordinateur de l'utilisateur, d'importer une image choisie. Lorsque l'utilisateur clique sur le bouton, la fenêtre de navigation dans ses dossiers apparaît, et il peut alors choisir n'importe quelle image présente sur son ordinateur. Une fois l'image sélectionnée, celle-ci s'affiche dans la fenêtre principale de l'application.

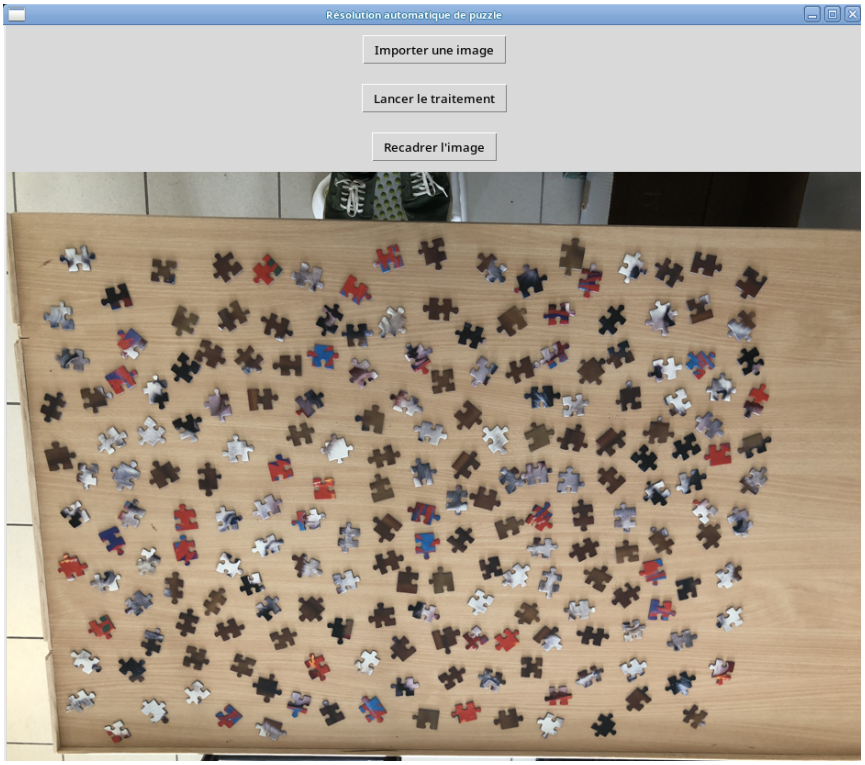


FIGURE 8 – Fenêtre principale de l'interface graphique avec une image chargée

7.1.2 Rogner des images

Le bouton se situant juste au dessus de l'image chargée, intitulé "recadrer l'image", permet de rogner l'image choisie selon une sélection faite par l'utilisateur. Pour cela, lorsque celui-ci clique sur le bouton, une nouvelle fenêtre s'ouvre avec l'image sélectionnée et l'utilisateur peut alors avec sa souris dessiner un rectangle, qui représentera la nouvelle sélection. Si l'utilisateur n'est pas satisfait de sa première sélection, il lui suffit de dessiner un nouveau rectangle directement dans la même fenêtre.

FIGURE 9 – Fenêtre de rognage de l'interface graphique avec une sélection de rognage effectuée

Une fois que l'utilisateur a fini de sélectionner la zone à rogner, la nouvelle image est sauvegardée dans le même répertoire. Pour réaliser la distinction entre les 2 images, le nom de l'image rognée a été modifié. Par exemple, si l'image de base sélectionnée par l'uilisateur portait le nom "myImage.jpg", la nouvelle image se nommera "myImage_cropped.jpg". Un message s'affiche sur le fenêtre principale de l'application pour signifier la sauvegarde de la nouvelle image.

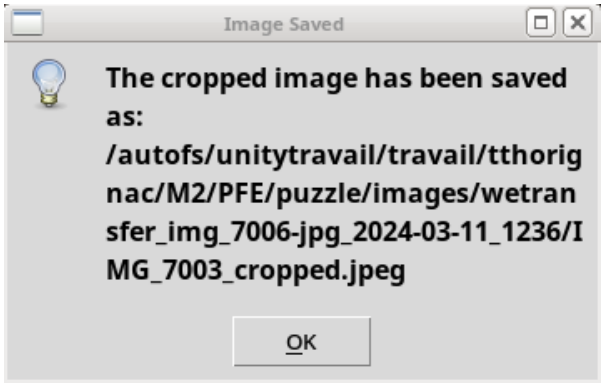


FIGURE 10 – Fenêtre principale de l'interface graphique avec le message de sauvegarde de la nouvelle image

L'utilisateur peut alors fermer la fenêtre dans laquelle il a recadré l'image, et peut charger la nouvelle image de la même manière que la précédente.

7.1.3 Segmentation

Une fois que l'utilisateur a une image qui lui convient, il peut lancer la segmentation de l'image en utilisant le dernier bouton "Lancer le traitement". Une animation de chargement apparaît alors dans la fenêtre principale pour signifier que la segmentation est en cours.

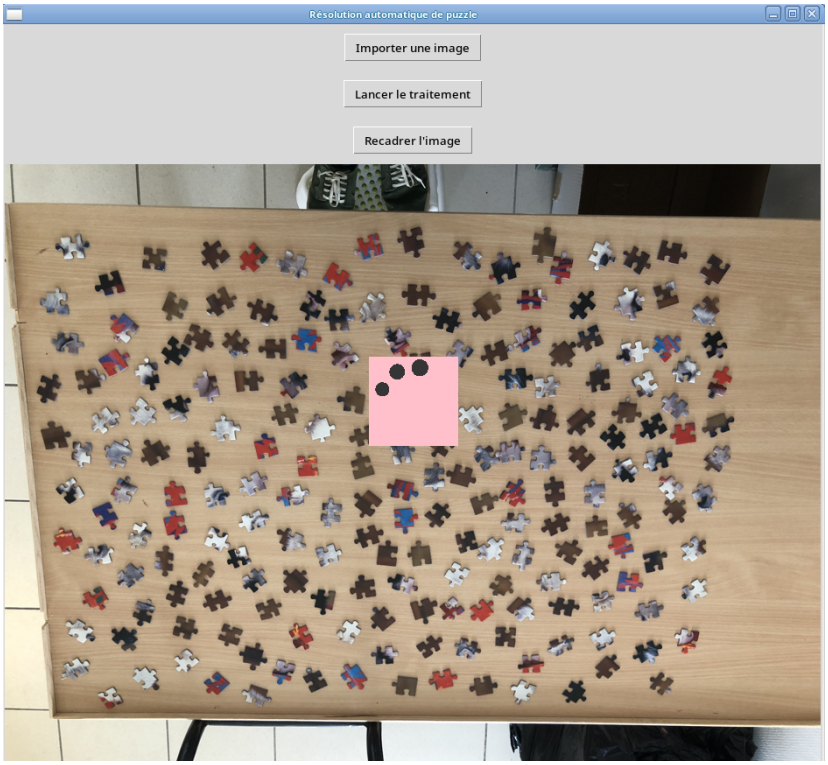


FIGURE 11 – Fenêtre principale de l'interface graphique avec l'animation de chargement

Une fois le traitement terminé, l'animation s'arrête et le résultat de la segmentation est ensuite traité et sauvegardé dans le fichier "result.png", dans le répertoire "Application".

8 Algorithmes/Méthodes/Données

8.1 Génération du *dataset* et entraînement du modèle par deep learning

La génération du *dataset* est une partie fondamentale de notre projet. En effet, il nous fallait créer un jeu de données suffisamment conséquent pour pouvoir entraîner notre modèle et ainsi pouvoir réussir à faire de la détection de pièces de puzzle sur des photos en réutilisant ce modèle.

Pour cela, nous avons choisi de générer artificiellement un grand nombre d'image différentes avec un nombre de pièces aléatoire. Le but est d'obtenir des images le plus réaliste possible afin que le modèle parvienne à reconnaître les pièces de puzzle sur une vraie photo.

Nous avons utilisé pour générer ces images un fond réaliste : nous avons demandé au client de nous fournir une photo de la table sur laquelle il souhaite réaliser l'acquisition des images comportant les pièces de puzzle pour que notre modèle puisse s'entraîner dessus.

La création de ces images s'est déroulée en plusieurs étapes : création des images de pièces vierges à partir de vraies pièces de puzzle noires, création des labels de ces pièces de puzzle à la main (Gimp), ajouts d'images sur les pièces de puzzle noires, implémentation de filtres pour diversifier les couleurs des pièces de puzzle, création des images du *dataset* en incrustant les pièces de puzzle sur un fond réaliste.

8.1.1 Modification des pièces de puzzle

1. Seuillage sur les photos des pièces de puzzle pour les délimiter et faire des labels

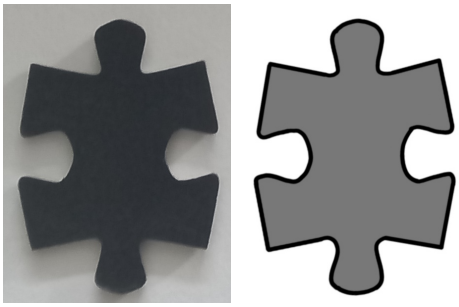


FIGURE 12 – Une pièce et son label associé (pièce/contour/fond)

2. Ajout d'image sur les pièces pour ajouter des motifs

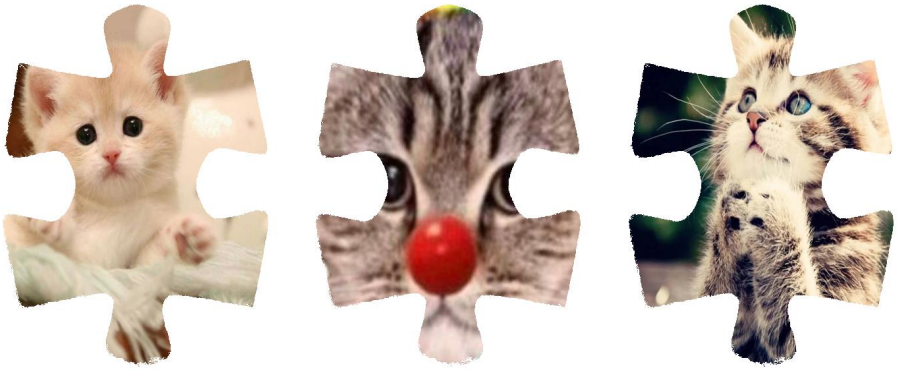


FIGURE 13 – Incrustation de différentes images sur des pièces

3. Ajout de filtres de couleurs et de modification de luminosité

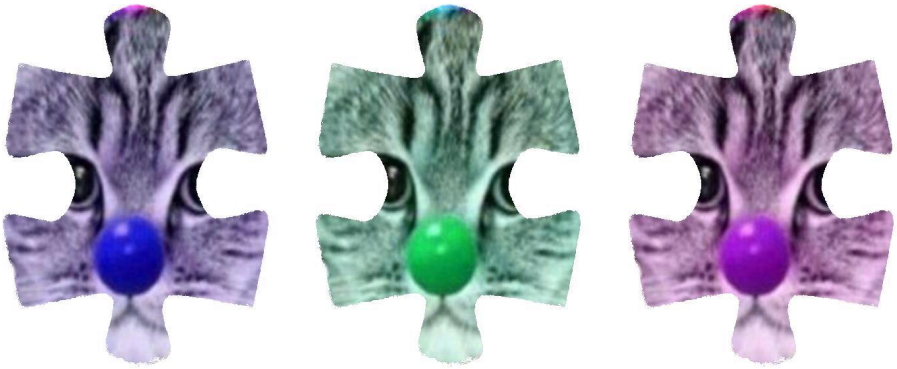


FIGURE 14 – Changement de chrominance/rotation de canaux sur les pièces

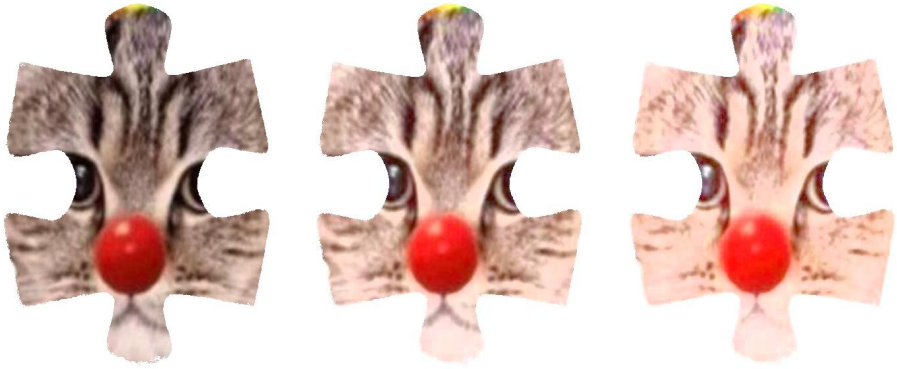


FIGURE 15 – Changement de luminance sur les pièces

8.1.1.1 Première passe

FIGURE 16 – Images générées et label associés

Les résultats avec nos images générer semblaient corrects, mais en essayant de segmenter avec différentes prises de vue réel ils n'étaient pas convainquant. Voici un exemple de ce que l'on pouvait obtenir :

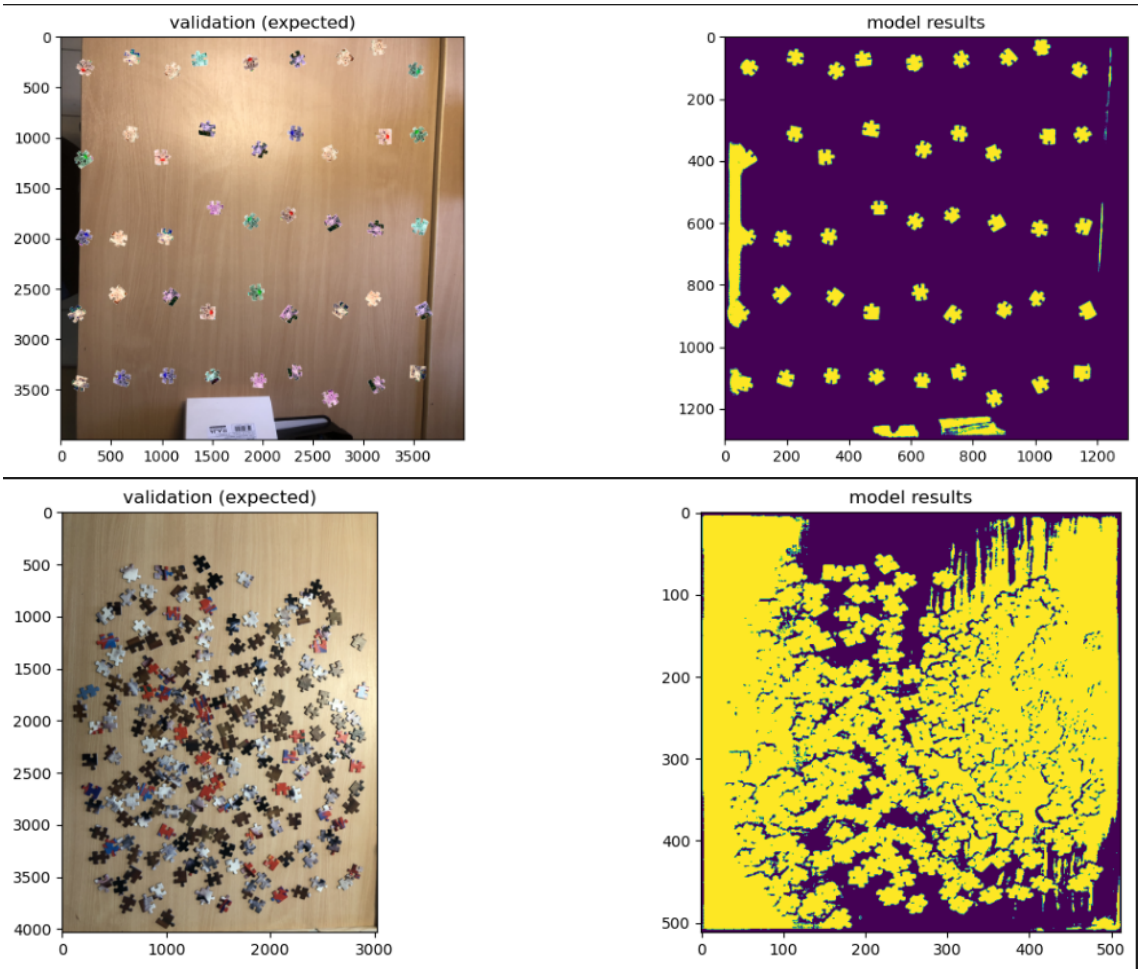


FIGURE 17 – Image générées/réel et résultats obtenus

Après différents tests afin d'identifier le problème, nous en sommes venus à la conclusion que le problème venait probablement de notre jeu de données. Par conséquent, nous avons choisi d'apporter quelques modifications à notre jeu de données afin de l'améliorer.

Nous avons d'abord modifier la photo utilisée pour l'arrière plan afin d'éviter de donner au modèle une photo de la table où les bords de celle-ci étaient présent. En effet, nous avons constaté à plusieurs reprises que les bords de la tables pouvaient être assimilés par le modèle comme des contours de pièce. Nous avons aussi apporté cette modification car des éléments autres que des pièces de puzzle étaient présents sur la table.

En plus de cela, nous avons ajouté des rotations aléatoires sur le fond, pour pour que le modèle ne fasse pas du sur-apprentissage sur un fond unique (nous avions également essayer de changer la couleur de la table de pseudo-aléatoire mais les premiers résultats avec cette méthode ne semblait pas prometteur nous n'avons donc pas continuer avec).

8.1.2 Ajout des textures et des effets de lumière

Nous avons ensuite pensé que les pièces de puzzle que nous avons créées artificiellement n'étaient pas assez réalistes. Nous avons donc changé les images utilisés pour qu'elles correspondent plus à des images que l'on pourrait retrouver sur des pièce de puzzle. Nous avons également choisi d'implémenter de nouveaux algorithmes de traitement d'images afin de tenter d'ajouter du réalisme aux pièces de puzzle en implémentant un algorithme ajoutant des textures aux pièces de puzzle.

FIGURE 18 – Pièces de puzzle avec des textures

Enfin, nous avons également constaté que les reflets de lumières présentes sur les photos posaient des problèmes lors de la segmentation. Les matériaux de la plupart des tables sont de telles sortes que sur la plupart des photos, on peut voir un spot lumineux précis provenant soit de la lumière ambiante de la pièce dans laquelle est prise la photo, soit du flash de la caméra. Pour faire en sorte que les images générées de notre *dataset* soit fidèles à cette particularité visuels, nous avons donc réalisé un algorithme qui rajoute de manière aléatoire des spots lumineux sur une image, avec une taille donnée :

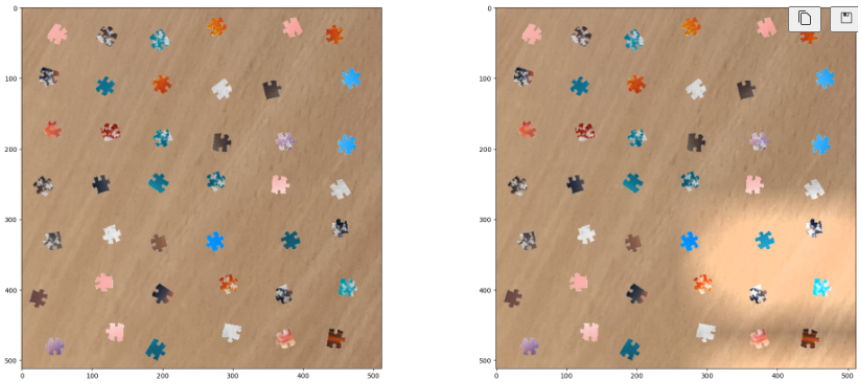


FIGURE 19 – Ajout d'éclairage artificiel sur une image générée

8.1.3 Dataset

Nous avons à partir de ces images de pièces de puzzle modifiées, créé artificiellement un jeu de données de 2000 images. Nous avons également, pour chaque image du *dataset*, généré les labels associés pour pouvoir entraîner le modèle. Nous avons choisi de générer des images de taille (512,512), car nous ne pouvions pas entraîner le modèle sur des images de plus grande taille.

Nous avons également testé dans un second temps le même *dataset* mais en passant les images en niveaux de gris pour analyser le comportement du modèle.

8.1.4 Entraînement du modèle

Nous avons choisi de faire des batchs de taille 6, sur des images de taille (512,512). Nous aurions voulu faire des batchs de plus grande taille, mais nous n'avons pas pu car l'ordinateur utilisé pour l'entraînement du modèle ne possédait pas une carte graphique assez puissante. Le matériel utilisé pour l'entraînement du modèle a clairement été un facteur limitant, que ce soit

pour le temps ou pour les données que nous avons choisi de générer pour le *dataset*.

Nous avons réalisé l'entraînement de ce modèle sur le matériel suivant : CPU i9-12900 16 coeurs, GPU RTX 3060 (12Go), RAM 64 Go.

Le modèle que nous avons obtenu et donnant les meilleurs résultats est un modèle entraîné sur des images passées en niveaux de gris.

8.2 Algorithmes

8.2.1 Déformation des images (perspective)

Afin d'augmenter le réalisme des images d'entraînement, nous avons souhaité pouvoir déformer les pièces synthétisées, de manière à adapter leur aspect à la perspective de la table (sur la photo de fond) sur laquelle nous souhaitons les incruster.

L'algorithme prendrait donc une image représentant des pièces de puzzle disposées aléatoirement, perpendiculaires à la caméra, et sans distorsion. Cette image représente des pièces que l'on aurait prises avec un caméra formant un angle de 90 degrés avec le support plat sur lequel elles seraient disposées, (ce qui, évidemment, ne sera pas toujours le cas lors d'une acquisition). Nous cherchons alors à transformer cette image de manière à ce que les coins de l'image coïncident avec les coins de la table sur la photo, en interpolant la transformation des points à l'intérieur.

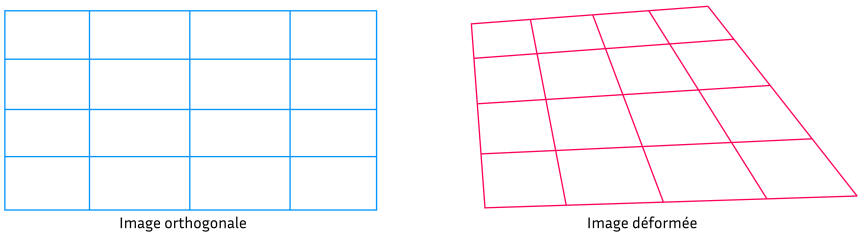


FIGURE 20 – Illustration de la déformation désirée

Dans un premier temps nous avons implémenté un modèle heuristique de déformation sur grille de points, en vue de l'adapter ensuite à un algorithme de déformation (heuristique) d'image.

Les données d'entrée sont les dimensions de l'image de base des pièces aléatoires, et les 4 points d'ancrage de la déformation.

La première étape que nous souhaitions franchir, était celle de déformer l'image en ne bougeant qu'un point, horizontalement ou verticalement, car

nous avons émit l'hypothèse que nous pouvons additionner ces transformations, afin d'obtenir la déformation suivant les deux axes ; puis, de continuer en ajoutant les transformations sur les autres points.
Voici alors notre point d'entrée :

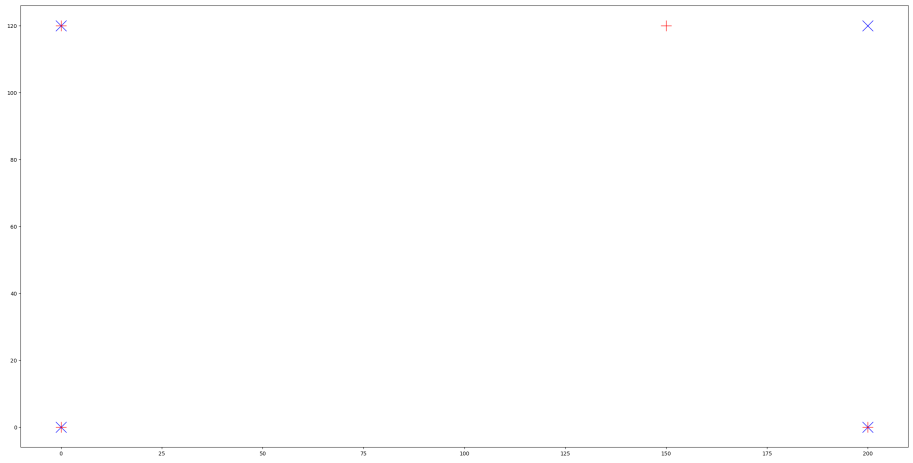


FIGURE 21 – Données d'entrée

Sur la Figure 12, les croix 'x' bleues représentent les dimensions de l'image des pièce de base, et les croix '+' rouges représentent les points de déformation.

Dans un premier temps, on cherche à interpoler un ensemble de points représentant des pixels de l'image :

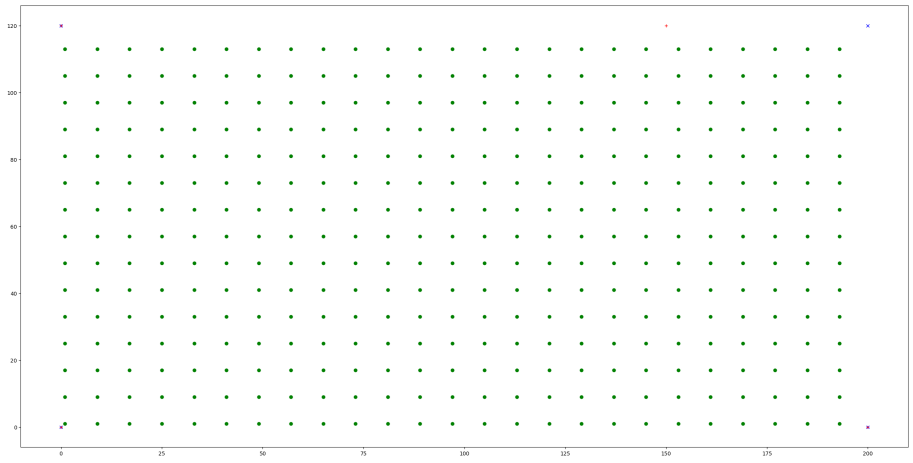


FIGURE 22 – Nuage de point à modifier

Les points seront ramenés sur des lignes "verticales" suivant la déformation des 4 points, ces lignes sont apparentes en bleu sur la Figure 14 :

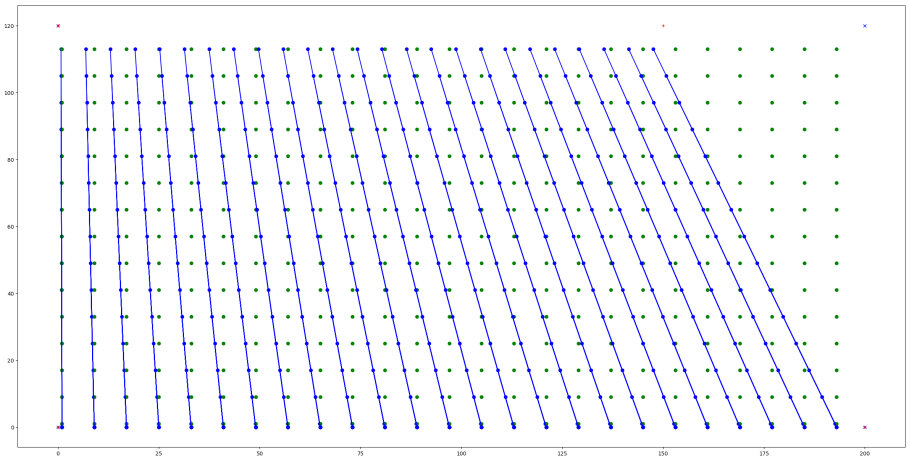


FIGURE 23 – Lignes "verticales" de perspective

Il faut ensuite trouver l'endroit où chaque point va s'insérer sur la ligne "verticale". Pour cela, on utilise des vecteurs, calculés en chaque point, qu'on peut visualiser sur la Figure 15. Ces vecteurs sont calculés selon la formule suivante $45 * p_x + 45 * p_y$, où 45 représente un demi-angle de 90 degrés, p_x et p_y l'endroit (en pourcent) de l'abscisse (resp. l'ordonnée) sur la largeur (resp. la hauteur) du point de l'image de base.
À ce calcul, nous avons rajouté et testé une adaptation de la valeur des pourcentages p_x et p_y, suivant une fonction mathématique continue et définie sur l'intervalle $[0, 1]$ et vers l'intervalle $[0, 1]$. Ce point sera détaillé à la fin de cette sous-section.
Sur la Figure 15, on peut visualiser les vecteurs (sans adaptation) :

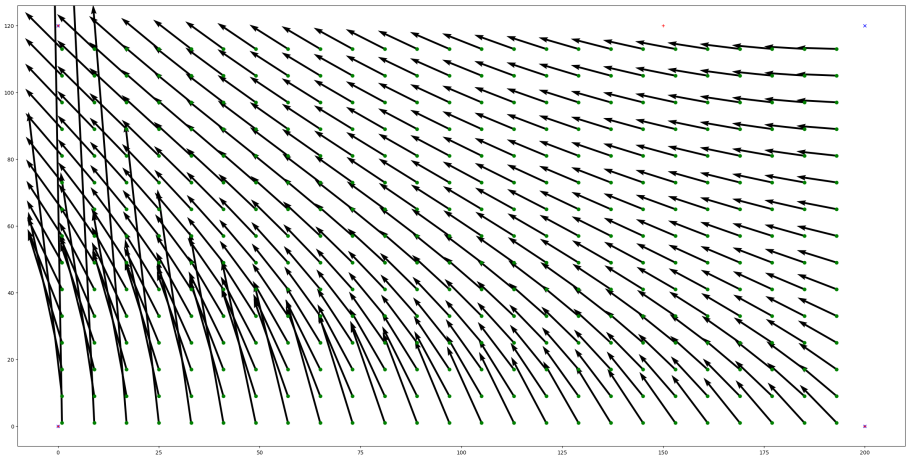


FIGURE 24 – Vecteurs de perspective

Ensuite nous cherchons l'intersection entre les lignes bleues et la droite qui correspond au vecteur passant par le point x de l'image dont on souhaite calculer la transformation. Nous obtenons des points "courbés", puis nous les rectifions en faisant une moyenne. Voici les résultats :

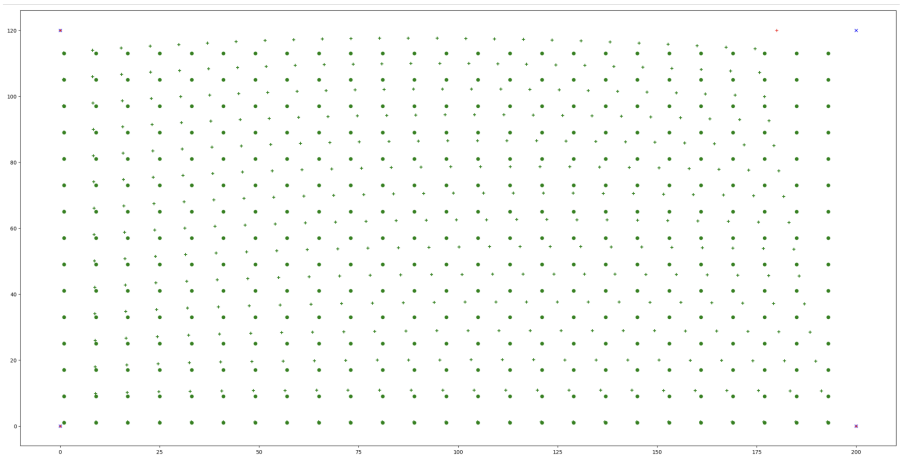


FIGURE 25 – Résultat de déformation, avec courbure

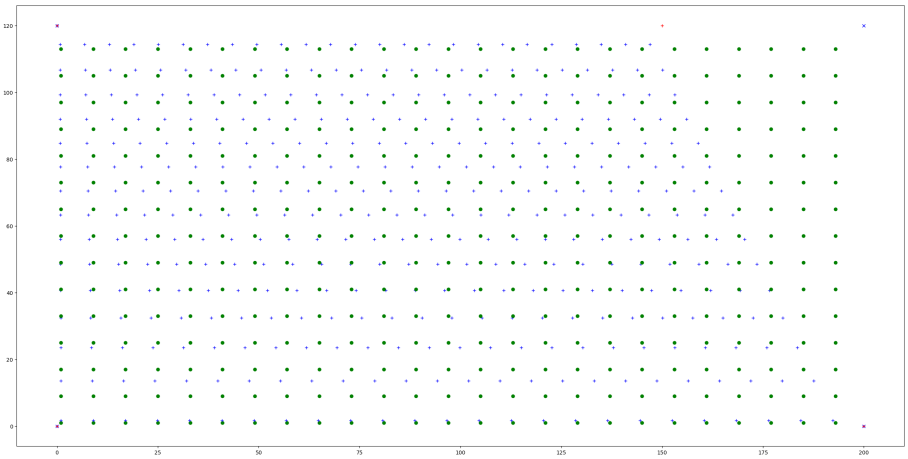


FIGURE 26 – Résultat de déformation, sans courbure

Ensuite, nous avons essayé avec une déformation plus forte, et avec tous les points à l'intérieur de l'image, voici une visualisation du résultat :

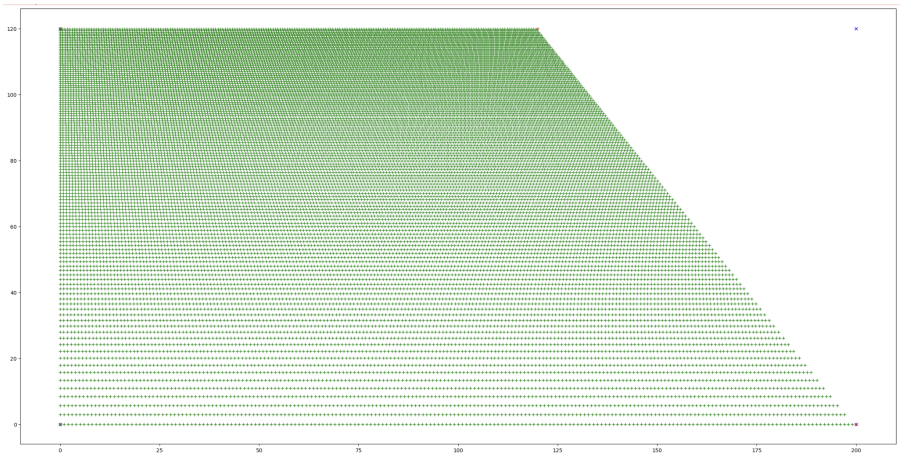


FIGURE 27 – Résultat de déformation, avec tous les points

Visuellement, les résultats semblent donner l'effet recherché. Les points "proches de la caméra" (en bas), sont bien "expandus", et les points "loins de la camera" (en haut) sont "contractés".

En ce qui concerne l'adaptation de p_x et p_y susmentionné, ça a pour effet, sur le résultat final, d'accentuer ou de réduire la disparité entre les points "proches de la caméra", mais nous avons mal compris comment cela fonctionne. L'effet dépend de la déformation (les 4 nouveaux points). Par exemple, en appliquant la fonction racine quatrième à p_y, on obtient une atténuation de cette disparité, représentée sur la Figure 19 :

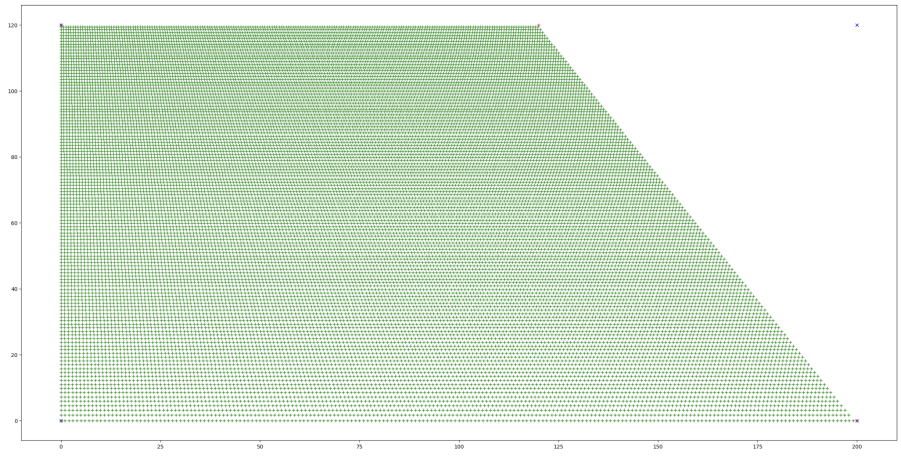


FIGURE 28 – Résultat de déformation, avec adaptation

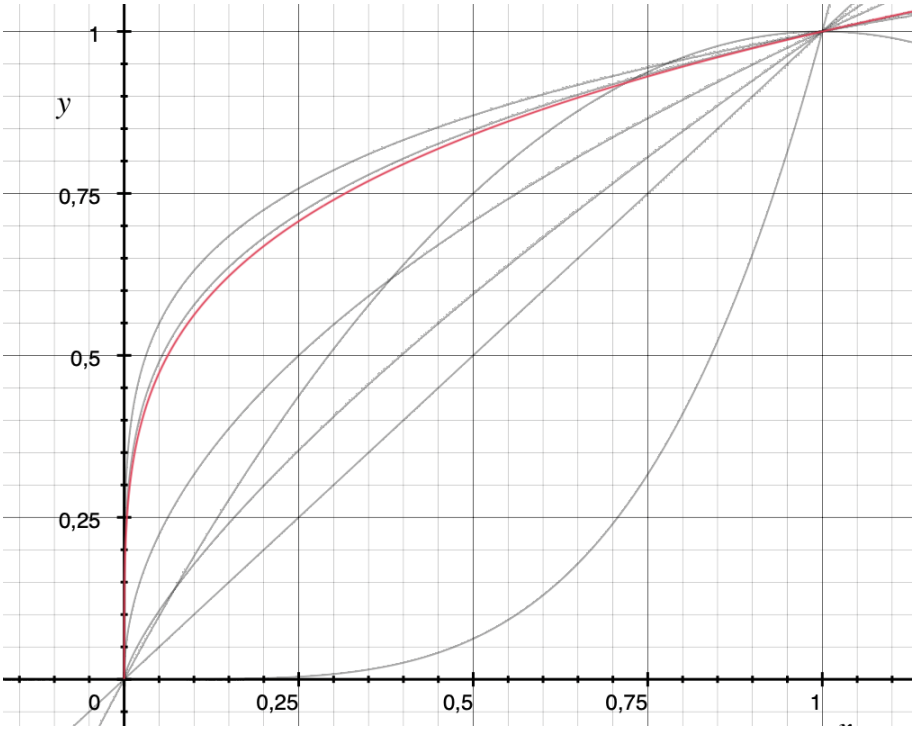


FIGURE 29 – Fonction $y = \sqrt{x}$ appliquée pour l'adaptation

Nous avons essayé toutes les fonctions de la Figure 20, avec différentes intensité de déformation. Cela nous a permit de corriger des erreurs lorsqu'on dépassait un certain seuil de déformation, par exemple quand le points de déformation allait devenait plus petit que la moitié de la largueur de l'image, cependant nous n'en avons pas dégagé quelque chose de clair, et nous n'avons pas besoin d'une déformation très importante pour notre projet.

Arrivé à ce stade de l'algorithme, nous avons pris conscience que le temps allait manquer pour avancer sur le reste du pipeline, et que les avantages qu'allaient nous apporter cet algorithme ne seraient pas assez importants pour le terminer dans l'immédiat. Nous l'avons alors laissé en tant que modèle à développer. L'étape suivante, c'est d'appliquer une méthode de *up sampling* sur les pixels qui sont "expandus", et de résoudre les conflits entre les pixels qui sont "contractés". Les conflits pourraient potentiellement être résolus par moyenne, ou par échantillonnage.

8.2.2 Dé-bruitage par morphologies mathématiques

Après entraînement du modèle, nous avons remarqué que les résultats de la segmentation comportent parfois du bruit, c'est-à-dire que le modèle

détecte des pièces où il n'y en a pas (à cause du fait que la table ne soit pas unicolore), et certaines pièces ne sont pas complètes. Afin de réduire le nombre de faux positifs et essayer de diminuer les erreurs dans la détection des pièces, nous avons appliqué à l'image résultante de la segmentation des traitements en utilisant des algorithmes de morphologies mathématiques.

Nous avons choisi d'utiliser comme morphologies mathématiques les fonctions de fermeture et d'ouverture de la bibliothèque `OpenCV`. Pour utiliser ces fonctions, il nous fallait choisir une taille de noyau pour réaliser la transformation. Nous nous sommes donc appuyés sur la fonction d'indice de similarité structurale (SSIM), en essayant différentes tailles de noyau et en conservant le meilleur score, par rapport à une image de segmentation que nous avons traité à la main afin de retirer les artefacts et combler au maximum les pièces. A la suite de l'utilisation de cette fonction, nous avons trouvé que nous obtenions les meilleurs résultats avec un noyau de taille (5,5) pour la fermeture et un noyau de taille (3,3) pour l'ouverture. On peut cependant se poser des questions quant à la pertinence de ce choix pour une image choisie quelconque, puisque nous n'avons réalisé un test que sur une image de segmentation. Par conséquent, cette taille de noyau pourrait s'avérer être variable en fonction de la taille des artefacts et des trous à combler. Plus généralement, nous pensons que la taille de noyau devrait varier en fonction de la taille des pièces présentes sur l'image, et donc dépend de la photo donnée en entrée.

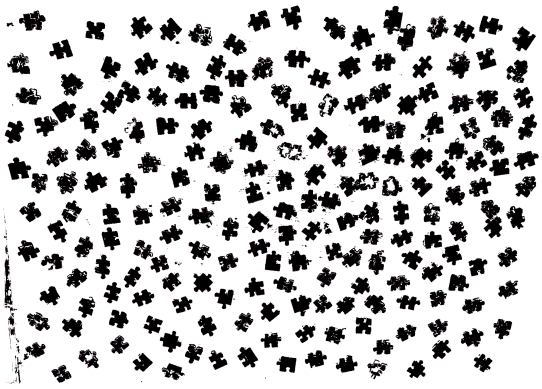


FIGURE 30 – Résultat de la segmentation non traitée

Cependant, même après utilisation des algorithmes de morphologies mathématiques, certaines pièces qui possédaient de trop grandes zones non détectées ou qui étaient seulement détectées que très partiellement n'ont pas pu être reconstruites de manière à être exploitable dans la suite du programme, qui réside en la reconstitution du puzzle.

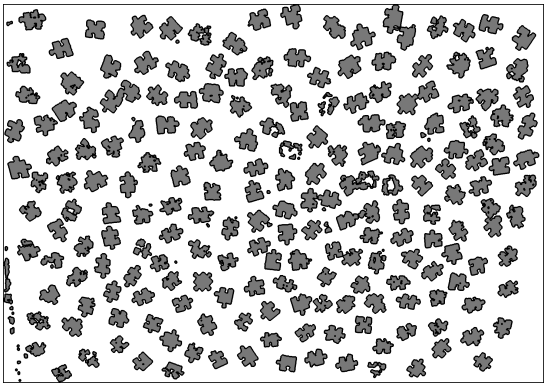


FIGURE 31 – Résultat de la segmentation traitée avec modification de couleur et ajout des contours

8.3 Séparation des pièces

La segmentation du modèle de réseau de neurones profonds, nous produit un label sous forme d'image, qui contient un grand nombre de pièces sur la même image. Nous avons besoin de les séparer et d'extraire chaque pièce, afin de les passer au solveur, pour vérifier que le puzzle est complet. Pour cela, nous avons souhaitions mettre au point un algorithme, dont le principe est assez simple. Lors d'un parcours classique de l'image, quand on rencontre un pixel noir (couleur à priori du contour de la pièce), on poursuit le parcours par adjacence, où le pixel suivant sera sélectionné si la moyenne des pixels autour de lui (obtenue avec un noyau de convolution) est de couleur minimale (la plus proche du noir possible). Il faut aussi contrôler la direction du parcours, pour éviter des cas où il ferait demi-tour alors qu'il ne faut pas, mais il faut pouvoir lui permettre de faire demi-tour lorsque cela est nécessaire.
Pour vérifier qu'on ne fait pas de changement de direction trop incohérent (comme un demi-tour brusque quand il ne faut pas), on regarde l'angle entre le vecteur de déplacement précédent, et le candidat au déplacement actuel. Le seuil (bas) est fixé arbitrairement, mais doit être fixé dynamiquement et en fonction de la pièce. Avec un angle arbitraire, l'algorithme fonctionne sur un certain nombre de pièces, mais il suffit que changer légèrement certains paramètres pour que le fonctionnement ne soit plus correct. Un angle seuil trop petit empêche la pièce de prendre certains virages du contour, et un angle seuil trop grand permet certains demi-tours non désirés. Le juste équilibre entre les deux, choisi arbitrairement, convient à beaucoup de cas, mais pas à tous.

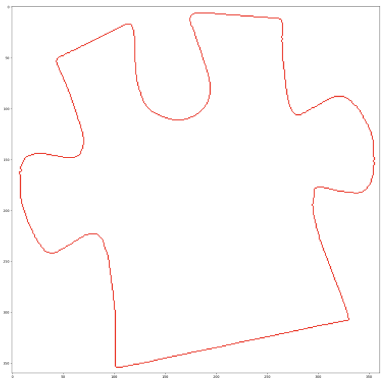


FIGURE 32 – Résultat de la séparation d'une pièce sur un label *GT* de plusieurs pièces

Une autre partie arbitraire du code, concerne la condition d'arrêt. On ne peut pas se contenter d'arrêter le parcours du contour (par adjacence) quand on retombe sur les pixels de départ, car au début il y a un "mini-calibrage", et il se peut que l'algorithme fasse un demi-tour, de plus, on n'a aucune garantie sur le fait qu'il n'a pas fait un demi-tour quelque part, et qu'il ne repasse pas par le point de départ avant de continuer sur le reste du contour normalement. Ce cas de détour ne pose pas de soucis sur la finalité de l'algorithme, car on cherche à isoler les pièces avec les bornes inférieures et supérieures sur les 2 axes de l'image.
L'avantage de cet algorithme, dans notre projet, comparé à un algorithme de contours classique et efficace, comme disponible dans la bibliothèque `OpenCV`, est que nous pouvons rajouter du code afin de permettre de faire des petites corrections sur les contours produits par le modèle de segmentation. Cet algorithme peut "continuer" si il rencontre un trou dans un contour. De plus, en jouant sur l'angle entre les vecteurs, il est envisageable de le faire évoluer de manière à lisser les éventuelles vaguelettes produites par le modèle sur le contour.

8.3.1 Solveur

8.3.1.1 Détection des creux et des excroissances des pièces

Pour différencier les pièces de bordures des autres pièces, on peut s'intéresser à leur géométrie. Les bordures hors coins ont 3 creux/excroissance, les coins en ont 2, et les autres pièces en ont 4. Ces aspects géométriques nous intéressent aussi pour vérifier que 2 pièces peuvent s'emboîter.

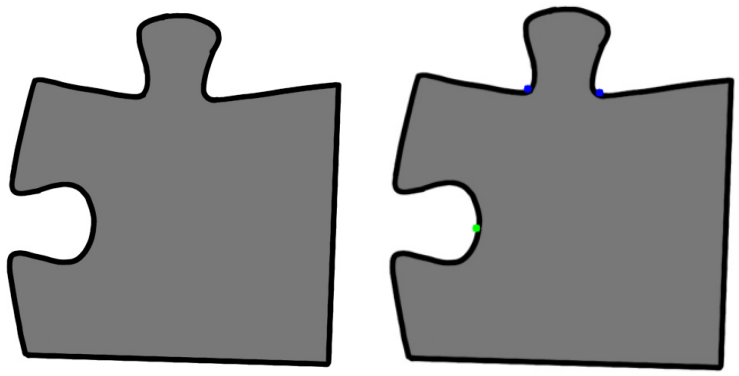


FIGURE 33 – Donnée d'entrée (à gauche) et détection des creux/excroissances (à droite)

On considère que l'intérieur de la pièce est blanche, et l'extérieur de la pièce est noir. Dans un premier temps on modifie les couleurs de l'image de manière à obtenir cette configuration sur les données d'entrée (dans l'exemple ci-dessus, ce traitement consiste à inverser les couleurs). L'image est convertie en niveau de gris, afin d'être traitée. `OpenCV` nous permet ensuite de récupérer les contours de la pièce, sous forme d'un ensemble de points, avec une méthode de seuillage. [5]

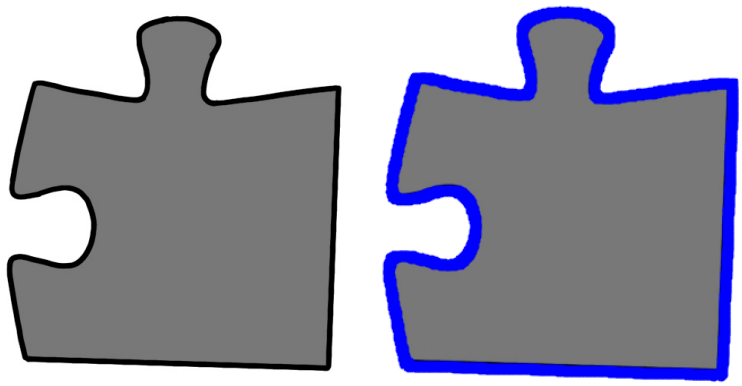


FIGURE 34 – Donnée d'entrée (à gauche) et détection du contour sous forme de nuage de points (à droite)

On récupère ensuite l'enveloppe convexe de ces points de contours, puis avec une autre méthode de seuillage on détermine si un défaut convexité trouvé correspond à un creux ou à une excroissance.

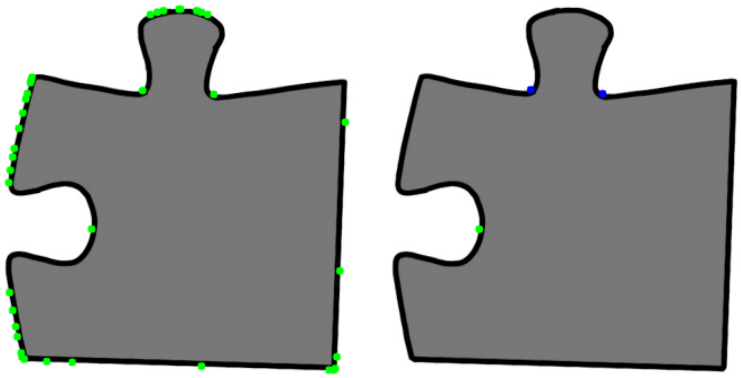


FIGURE 35 – Défauts de convexité (à gauche) et détection des défauts de convexité associés aux creux/excroissances par seuillage (à droite)

8.3.1.2 Détection des coins des pièces

Afin de déterminer si 2 pièces peuvent s'emboîter, nous avons besoin de regarder si les coins des pièces coïncident.

Dans un premier temps, nous avons voulu exploiter une idée, qui consiste à utiliser la régression linéaire, afin de repérer les zones qui sont courbées. Nous avons effectué des premiers tests.

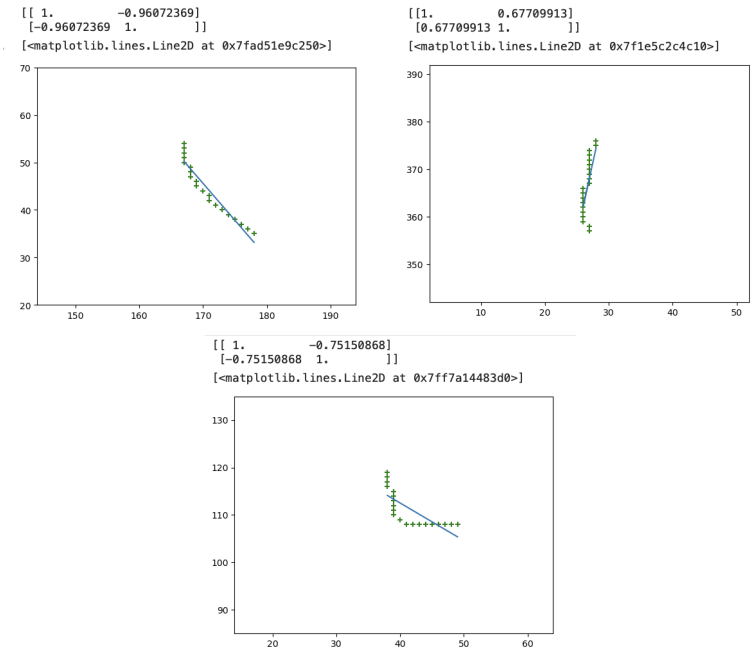


FIGURE 36 – Premiers résultats de la régression linéaire

Après avoir creusé cette piste, nous en avons déduit, que l'avantage de cette méthode avec régression linéaire, est de repérer les courbures des pièces, mais les inconvénients sont multiples : nous n'arrivons pas à différencier les courbures plus pointues, des courbures moins pointues, et on pense que c'est à cause d'imprécisions des scores, selon le nombre de pixels et l'inclinaison de la droite de régression. Par exemple, sur la Figure 28, on peut voir que les deux schémas en haut, que les courbes de points, marqués par des croix vertes, sont assez similaires en terme de courbure, pourtant, les scores absolus sont relativement différents. Nous pensons que cela est dû au fait que la droite de régression du schéma de droite (avec un score d'environ 0.67) devrait être verticale, et les modèle de régression que nous avons testés ne prennent pas cela en compte. Nous avons pu aisément écarter ce genre de cas, car en vue du ratio entre la hauteur et la largeur des points, on sait que ce n'est pas courbé.

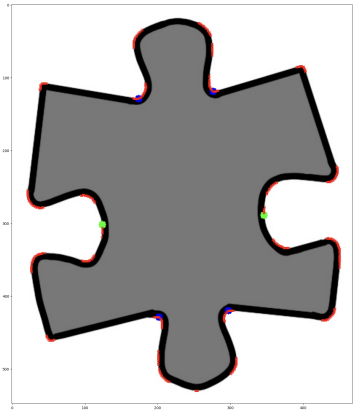


FIGURE 37 – Résultats finaux que l'on a obtenu avec la régression linéaire

Nous avons ensuite essayé d'exploiter le coefficient directeur de la droite, pour observer les moments où la dérivée est plus accentuée, mais nous nous sommes rendus à l'évidence que cette méthode ne conviendrait pas, car sur un coin plus arrondi comme le coin inférieur droit sur la Figure 29, on ne pourrait peut-être pas faire la différence.

Nous avons alors changé de méthode en cours de route. La nouvelle méthode part de l'hypothèse que l'on a une image de pièce orientée droite, comme sur la Figure 29. À partir de ce point de départ, après avoir centré la pièce au milieu, on trace deux droites orthogonales, qui passent le plus proche possible des 4 coins de l'image. Ensuite on fait un calcul d'intersections entre ces droites et le nuage de point qui représente le contour de la pièce, avec un epsilon (car le contour n'est pas continu), on extrait 4 intersections, 2 pour chaque droite. À partir de ces intersections, va procéder à des rotations de la droite, avec pour objectif de maximiser la distance entre les 2 points.

Cet algorithme fonctionne plutôt bien sur des coins pointus, mais en ce qui concerne les coins arrondis, il y a toujours une forte imprécision.

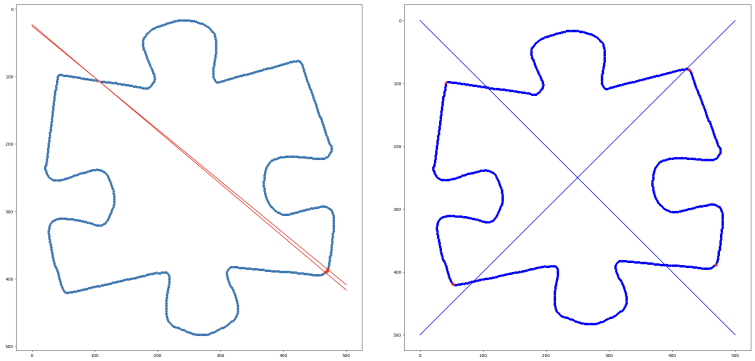


FIGURE 38 – Détection du coin inférieur droit par rotation depuis l'intersection supérieure gauche (*plot* de gauche), résultat final (*plot* de droite).

8.3.1.3 Emboîtement des pièces

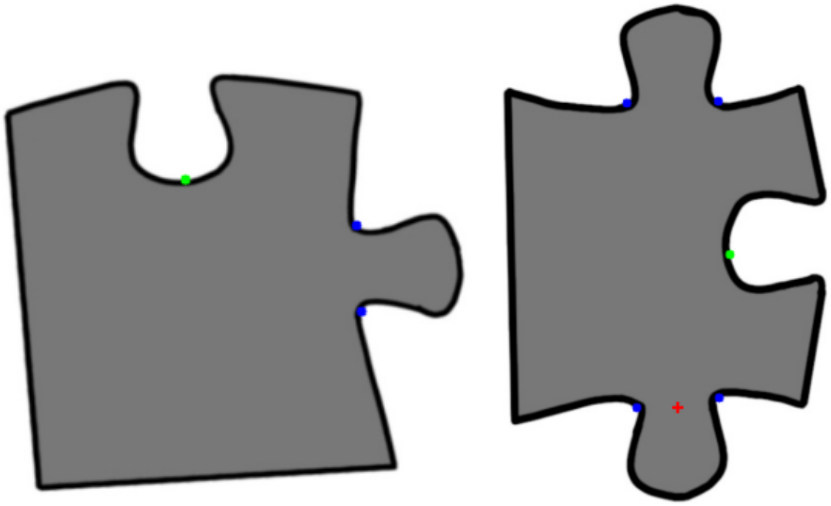


FIGURE 39 – Un coin(la pièce)(à gauche) et le bord à ajouter par dessus (à droite)

Pour savoir si nous pouvons emboîter deux pièces de puzzle ensemble comme les deux sur l'image ci dessus, tout d'abord on veut savoir si la pièce de droite est bien un bord et si oui est ce qu'on a le bord aligné avec celui du coin de la pièce de gauche si c'est le cas on cherche à savoir si le point rouge de la pièce de droite est aligné sur le même axe que le creux de la pièce de gauche qui est le point vert.

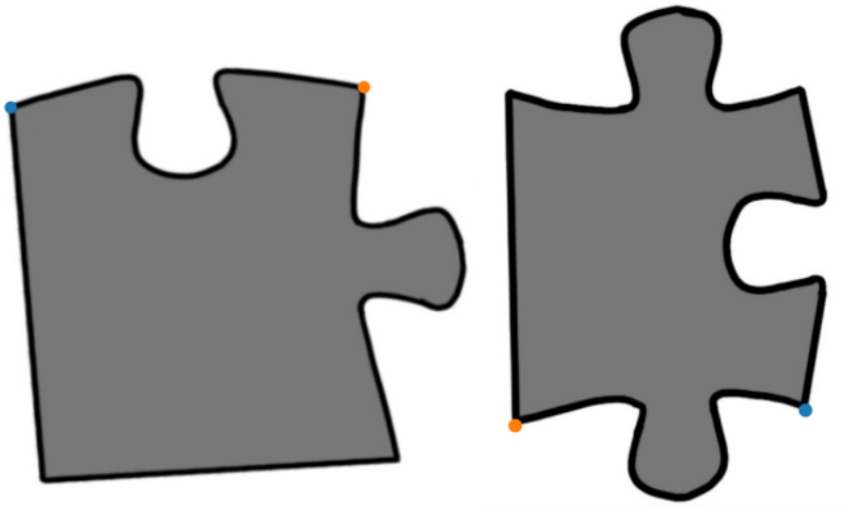


FIGURE 40 – Un coin (la pièce) et emplacement des deux coins (à gauche) et le bord à ajouter par dessus (emplacement des deux coins) (à droite)

Maintenant que l'on sait si la pièce de droite peut être dans le creux de la pièce de gauche si c'est le cas on vérifie si le coin orange de la pièce de droite est aligné sur le même axe que coin bleu de la pièce de gauche. Ensuite si le coin bleu de la pièce de droite est aligné sur le même axe que le coin orange de la pièce de gauche alors les deux pièces de puzzle peuvent s'emboîter.

8.3.1.4 Résolution du puzzle (au stade de conceptualisation)

Les algorithmes évoqués dans cette section sont au stade de la conceptualisation, ils n'ont pas été implémentés.

Les données d'entrée que l'on dispose pour résoudre le puzzle sont variables. On possède un nombre de pièces approximatif, une (ou des) photo(s) de toutes les pièces de la boite, certains fabricants fournissent les dimensions du puzzle, on peut estimer la taille et les dimensions moyens d'une pièce, et on a une image partiellement complète du puzzle final sur la boite (il y a souvent des obstructions par des logos).

Les seules données d'entrée dont on dispose toujours sont la (ou les) photo(s) des pièces de puzzle à l'intérieur de la boite. Un algorithme naïf et exhaustif à partir de ça, serait dans un premier temps de chercher les 4 coins, et de renvoyer faux si on n'en trouve pas exactement 4, puis, de récupérer tous les bords, de vérifier qu'ils peuvent former un rectangle en renvoyant faux si leur nombre est impair, puis, d'essayer toutes les combinaisons possibles d'emboîtement des pièces, avec toutes les dimensions possibles

de puzzle final. C'est un problème très combinatoire, à titre d'exemple, pour un Puzzle qui contient x pièces, à supposer que l'on a déjà placé la bordure complète de taille y, il faut tester quelque chose de l'ordre de $(x - y)!$ test d'emboîtements possibles au cours de la résolution du puzzle pour le remplir, et nous souhaitons résoudre des puzzles de plus de 100 pièces, ce qui fait un nombre d'opérations colossal.

Nous avons conceptualisé plusieurs optimisations possibles afin de réduire le nombre d'opérations.
Dans les cas où nous disposons des dimensions du puzzle, on peut estimer la moyenne de la dimension et de la taille d'une pièce de puzzle, et estimer le nombre de pièces que l'on doit avoir en bordure, respectivement en hauteur et en largeur.
Une autre optimisation que l'on souhaitait (prévisionnellement) mettre en place, était d'exploiter la segmentation des pièces face image, pour récupérer cette image, la passer sur la photo de l'image finale, déterminer un score de probabilité de présence de la pièce sur une zone de cette image, et s'en servir pour pré-attribuer une zone réduite à chaque pièce du puzzle.
Enfin, de manière plus globale, nous avons pensé à tenter de convertir le problème en une formule SAT. Les variables de la formule seraient les pièces, à chacune de leur position possible dans le puzzle, soit la forme $x_{i,j}$ où i est le numéro correspondant à une pièce et j le numéro correspondant à l'emplacement de cette pièce. Une variable $x_{i,j}$ est vraie si la pièce i est placée à l'emplacement j. La formule devrait avoir x variables valuées à vrai, et les autres à faux, de manière à ce que les i, et respectivement les j, de chaque variable soient tous différents les uns des autres. De ce fait, chaque pièce serait placée à un seul endroit et tous les endroits seraient occupés par une pièce différente. Pour construire la formule, on devrait utiliser la fonction qui permet de savoir si 2 pièces s'emboîtent, et exploiter le fait que si une pièce est un coin, elle doit s'emboîter avec 2 pièces, si c'est une bordure avec 3 pièces, et toutes les autres pièces doivent s'emboîter avec 4 pièces. Pour exprimer le fait qu'une pièce, par exemple de coin, s'emboîte avec 2 pièces, à la position k, où ses cases adjacentes orthogonalement sont à la position k_a et k_b, on peut utiliser la formule suivante : $\bigvee_{i \neq i_1 \neq i_2} \texttt{emboite}(x_{i,k}, x_{i_1,k_a}) \wedge \texttt{emboite}(x_{i,k}, x_{i_2,k_b})$, où `emboîte(p1, p2)` est la fonction qui permet de savoir si 2 pièces données s'emboîtent. Pour vérifier qu'on ne met pas deux fois la même pièce au même emplacement k, on pourrait rajouter le terme $\bigwedge_{i_1 \neq i_2} \overline{x_{i_1,k} \wedge x_{i_2,k}}$ à la formule globale. On peut utiliser la même logique pour les bords et les autres pièces.
Il reste à savoir si le fait qu'on ne se soucie pas de l'orientation des emboîtements ne pose pas problème. Par exemple, on pourrait se retrouver avec un emboîtement d'une pièce par 4 pièces comme ce serait requis, mais avec 3 pièces qui s'emboîtent sur une seule de ses arrêtes. On suppose qu'il n'existe

pas beaucoup de combinaisons possibles qui satisfassent la formule SAT, et il serait possible éventuellement de demander une vérification humaine à partir des résultats visuels fournis par la résolution de cette formule. Nous pensons qu'il y aura plusieurs combinaisons possibles, car si 2 pièces ont la même forme, elles peuvent être permutées.

9 Tests

Ayant accumulé du retard sur plusieurs points importants du projet au cours de sa réalisation, nous n'avons pas pris le temps de réaliser des tests sur les différents éléments du projet, mis à part quelques tests visuels sur les résultats obtenus, puisque nous avons travaillé sur des images tout au long du projet.

9.1 Tests du réseau neuronal convolutif (UNet)

Pour les tests concernant le modèle, étant donné que nous avons utilisé un modèle pour de la segmentation d'image, la grande majorité des tests à effectuer sont de la vérification visuel, que nous avons donc effectuer à la main.

10 Projet : comparaison Gantt prévisionnel/effectif

10.1 Gantt effectif

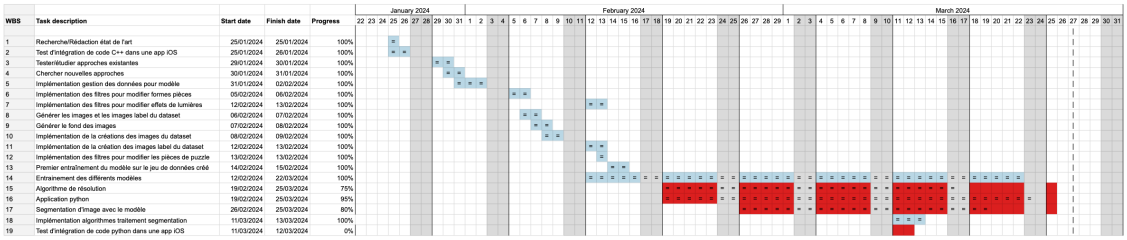
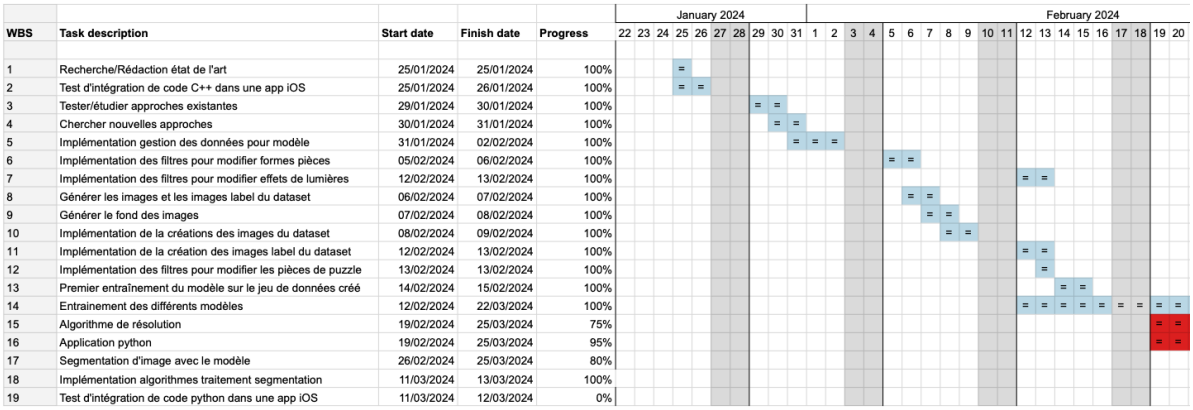


FIGURE 41 – Gantt effectif complet



WBS	Task description	Start date	Finish date	Progress
1	Recherche/Rédaction état de l'art	25/01/2024	25/01/2024	100%
2	Test d'intégration de code C++ dans une app iOS	25/01/2024	26/01/2024	100%
3	Tester/étudier approches existantes	29/01/2024	30/01/2024	100%
4	Chercher nouvelles approches	30/01/2024	31/01/2024	100%
5	Implémentation gestion des données pour modèle	31/01/2024	02/02/2024	100%
6	Implémentation des filtres pour modifier formes pièces	05/02/2024	06/02/2024	100%
7	Implémentation des filtres pour modifier effets de lumières	12/02/2024	13/02/2024	100%
8	Générer les images et les images label du dataset	06/02/2024	07/02/2024	100%
9	Générer le fond des images	07/02/2024	08/02/2024	100%
10	Implémentation de la créations des images du dataset	08/02/2024	09/02/2024	100%
11	Implémentation de la création des images label du dataset	12/02/2024	13/02/2024	100%
12	Implémentation des filtres pour modifier les pièces de puzzle	13/02/2024	13/02/2024	100%
13	Premier entraînement du modèle sur le jeu de données créé	14/02/2024	15/02/2024	100%
14	Entrainement des différents modèles	12/02/2024	22/03/2024	100%
15	Algorithme de résolution	19/02/2024	25/03/2024	75%
16	Application python	19/02/2024	25/03/2024	95%
17	Segmentation d'image avec le modèle	26/02/2024	25/03/2024	80%
18	Implémentation algorithmes traitement segmentation	11/03/2024	13/03/2024	100%
19	Test d'intégration de code python dans une app iOS	11/03/2024	12/03/2024	0%

FIGURE 42 – Gantt effectif partie gauche

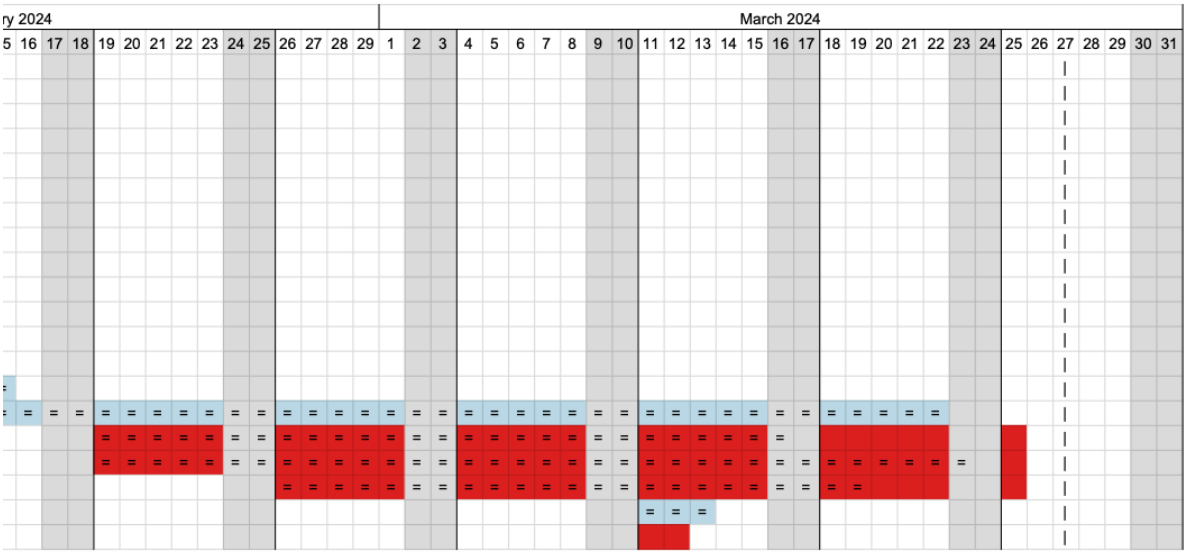


FIGURE 43 – Gantt effectif partie droite

Lorsque l'on compare notre Gantt prévisionnel à notre Gantt effectif, nous pouvons constater que certaines tâches non prévues ont dû être ajoutées à notre projet, ce qui a entraîné du retard de certaines tâches. D'autres étapes ont pris également du retard à cause de problèmes d'implémentation difficiles à résoudre.

10.2 (non) Réalisation d'une application iOS

La réalisation des *features* du projet (modèle de segmentation UNet pour la détection des contours, séparation des pièces, assemblage des pièces,...) nous a pris plus de temps que nous ne l'avions envisagé initialement. De plus, lorsque nous avons essayé d'intégrer notre code Python à une application iOS, nous avons eu des complications.

Nous avions une référence à un dépôt GitHub fournie dans le sujet, qui contient un *solveur* de puzzle. Nous comptions l'utiliser pour notre projet, afin de nous concentrer sur la tâche difficile de la segmentation, avant de nous rendre compte qu'il n'était pas *open source*. Pour accueillir ce code dans le projet final, nous avions mit en place ce que la société Apple appelle l'interopérabilité entre C++ et Objective-C (*C++ and Objective-C Interoperability*) et qui permet de lier du code `C++` et du code `Swift`. Le code `Swift` nous permet de créer une interface utilisateur sur iOS, et le code C++ de résoudre les Puzzles à partir d'images nettes et rectifiées des pièces séparées.

Swift est un langage performant et intuitif proposé par Apple pour développer des applications iOS. Swift étant un langage jeune, beaucoup de bibliothèques et de base de code sont (et ne sont que) disponibles dans d'autres langages, il est alors intéressant de trouver des méthodes pour lier aisément Swift avec d'autres langages.

Plusieurs méthodes existent pour lier du code C++ à une application iOS en Swift. Récemment (en 2023), Apple a facilité l'interopérabilité entre Swift et C++ avec `SwiftUI`.
Sans `bridging-header` : Cette méthode est présentée par Apple dans une WWDC, des exemples de code sont disponibles en téléchargement. Après plusieurs essais, nous avons laissé tomber cette méthode. Les exemples que nous avons trouvé sont codés pour MacOS et pour iOS.
Avec `bridging-header` : Cette méthode est proche de la méthode précédente mais moins automatique. Il faut créer un `bridging-header`, qui est un fichier répertoriant les en-têtes des fichiers que l'on veut importer dans Swift.

Nous avons finalement obtenu un point de départ.

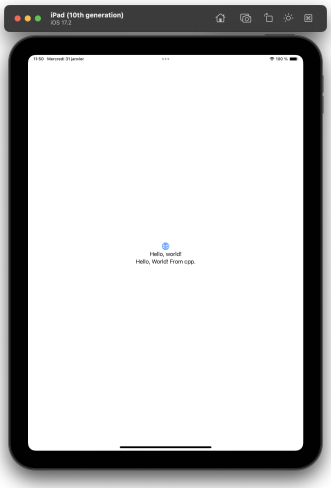


FIGURE 44 – Liaisons C++ avec une app Swift sur simulateur iPad

Nous avons d'abord voulu prototyper nos algorithmes, avant de les écrire en C++ pour les intégrer à une application iOS, alors nous avons continué notre travail en Python. Cependant, le prototypage était plus compliqué que nous l'avions envisagé, et nous sommes arrivés proches de la fin de l'échéance. Nous avons alors essayé de lier du code Python à du code Swift, mais c'est beaucoup plus compliqué que avec du C++. Après 2 journées de recherche et d'essais qui nous ont menés à la conclusion que cela prendrait beaucoup plus de temps, nous avons alors décidé de nous concentrer sur notre prototypage, et de faire une interface graphique sur ordinateur.

Conclusion (Bilan / Perspectives)

En utilisant l'intelligence artificielle, nous avons pu aborder de manière innovante la problématique de vérification de la complétude des puzzles, qui était jusque là abordé seulement par des outils de traitement d'image tel que `OpenCV`. Nous avons en utilisant un réseau neuronal convolutif essayé de rendre l'expérience utilisateur plus facile pour les personnes qui ne sont pas familières avec l'informatique.

Il est important de noter que notre application est seulement un prototype fait dans un cadre pédagogique et que, faute de temps et de prise en charge par un enseignant référent, nous n'avons pas pu réaliser tous ce que nous voulions faire.

Il existe des possibilités d'amélioration, notamment en ce qui concerne la

qualité et la quantité des données utilisées pour l'entraîner. Une meilleure diversité des puzzles et une augmentation de la taille de l'ensemble de données pourraient contribuer à une meilleure précision et à une plus grande robustesse de notre solution.

Il est également essentiel de souligner que même avec les progrès de l'IA, il est peu probable d'atteindre une précision parfaite. Les puzzles peuvent présenter des variations importantes en termes de nombre de pièces, de formes et de couleurs, ce qui rend la tâche de reconstruction automatique parfois complexe.

Il est également possible d'améliorer notre algorithme de résolution des pièces de puzzle en prenant en compte les couleurs des pièces de puzzle pour savoir si deux pièces vont bien s'emboîter ensemble car pour le moment nous avons juste par rapport aux contours des pièces.

Il est aussi possible d'améliorer l'algorithme en prenant en compte la résolution des pièces puisque l'acquisition des pièces se fait à partir d'une photo toutes les pièces n'auront pas la même résolution sur une même photo ou bien entre deux photos de puzzle différent.

Il serait possible d'optimiser continuellement notre outil, en explorant de nouvelles techniques d'apprentissage automatique et en affinant nos algorithmes pour obtenir des résultats encore meilleurs.

Références

[1] R Albertazzi. Solving jigsaw puzzles with python and opencv.
https://towardsdatascience.com/solving-jigsaw-puzzles-with-python-and-opencv-d775ba730660.
Accessed : 2024-03-25.

[2] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60 :91–110, 2004.

[3] Matthew Aguiar. *Jigsaw Puzzle Solver*. PhD thesis, WORCESTER POLYTECHNIC INSTITUTE, 2022.

[4] Pytorch-unet.
https://github.com/milesial/Pytorch-UNet.
Accessed : 2024-03-27.

[5] Convexity defects opencv.
https://theailearner.com/tag/convex-hull-opencv/.
Accessed : 2024-03-25.